C++ FOR C# DEVELOPERS

JACKSON DUNSTAN

JacksonDunstan.com
Copyright © 2021 Jackson Dunstan
All rights reserved

Table of Contents

1. Introduction

History

C++'s predecessor is C, which debuted in 1972. It is still the most used language with C++ in fourth place and C# in fifth.

C++ got started with the name "C with Classes" in 1979. The name C++ came later in 1982. The original C++ compiler, Cfront, output C source files which were then compiled to machine code. That compiler has long since been replaced and modern compilers all compile C++ directly to machine code.

Major additions to the language were added with "C++ 2.0" in 1989 and the language was then standardized by ISO in 1998. Colloquially, this was called C++98 and began the convention where the year is added to name a version of the language. It also formalized the process of designing and standardizing the language via a committee and various working groups.

Minor changes to the language in 2003 resulted in C++03, but the "Modern C++" era began with *huge* changes to the language in C++11. This also quickened the standardization process from the previous eight year gap to just three years. This meant that we got minor changes in C++14, relatively big changes in C++17, and *huge* changes once again in C++20. Game engines such as Unreal, Cryengine, and Lumberyard all support at least C++17, if not C++20.

At this point the language has little resemblance to C. Much C code will still compile as C++, but idiomatic C++ is only superficially similar to C.

Standard Library

Every release of C++ includes what is called the “Standard Library.” This is often called the “STL,” meaning “Standard Template Library,” for its heavy use of a C++ language feature called templates. This library is also standardized by ISO along with the language itself.

The Standard Library is similar to .NET’s Framework Class Library or CoreFX. The architectural approach is for the C++ language to have powerful, low-level language features so more can be implemented in libraries instead of directly included in the language. For example, the language doesn’t include a string class. Instead, the Standard Library provides a `string` class that is efficiently implemented with low-level language features.

The following table shows the major sections of the Standard Library and their loose equivalents in .NET:

Standard Library Section	**C++**	**C#**
Language support	`numeric_limits::max`	`int.MaxValue`
Concepts	`default_initializable`	`where T : new()`
Diagnostics	`exception`	`System.Exception`
Utilities	`tuple<int, float>`	`(int, float)`
Strings	`string`	`System.String`
Containers	`vector`	`List`
Iterators	`begin()`	`GetEnumerator()`

Standard Library Section	**C++**	**C#**
Ranges	`views::filter`	`Enumerable.Where`
Algorithms	`transform`	`Enumerable.Select`
Numerics	`accumulate`	`Enumerable.Aggregate`
Localization	`toupper`	`Char.ToUpper`
I/O	`fstream`	`FileStream`
File system	`copy`	`File.Copy`
Regular expressions	`regex`	`Regex`
Atomic operations	`atomic++`	`Interlocked.Increment`
Threading	`thread`	`Thread`

Some game programming environments do not use the Standard Library, or at least minimize its use. EA has implemented their own version called EASTL. Unreal has many built-in similar types (`FString` vs. `string`) and functions (`MakeUnique` vs. `make_unique`). These libraries benefit from the same low-level language features that the Standard Library is built on, but instead use them to efficiently reimplement what would be language features in many other languages.

Tools

The main tool is, of course, the compiler. There are many good options these days, but here are some of the most popular ones:

Compiler	**Cost**	**Open Source**	**Platforms**
Microsoft Visual Studio	Free and Paid	No	Windows
GCC (GNU Compiler Collection)	Free	Yes	Windows, macOS, Linux
Clang	Free	Yes	Windows, macOS, Linux
Intel C++	Free	No	Windows, macOS, Linux

There are also many IDEs available with the usual combination of features: a text editor, compiler execution, interactive debugger, etc. Here are some popular options:

IDE	**Cost**	**Open Source**	**Platforms**
Microsoft Visual Studio	Free and Paid	No	Windows
Apple Xcode	Free	No	macOS
JetBrains CLion	Paid	No	Windows, macOS, Linux

IDE	Cost	Open Source	Platforms
Microsoft Visual Studio Code	Free	Yes	Windows, macOS, Linux

Many static analyzers, known as “linters,” and dynamic analyzers are available. The Clang sanitizers suite is free, open source, and has Unreal support. Commercial tools such as Coverity SAST are also available. Clang format and many IDEs can enforce style guides and automatically reformat code.

Documentation

The C++ standard is available for purchase, but almost no C++ developers actually buy it. A draft version is available for free and will be nearly identical, but it is extremely long and technical so it is also only a reference of last resort. Instead of the standard itself, most developers read reference sites such as cppreference.com just as they would read Microsoft Docs (a.k.a. MSDN) for C# reference.

Many guideline documents exist for C++. The C++ Core Guidelines, Google C++ style guide, and engine-specific standards are all commonly used. The C++ Core Guidelines, in particular, has a companion Guidelines Support Library (GSL) to enforce and facilitate the guidelines.

Community

There are many places where the community of developers congregate. Here are a few:

- The C++ language Slack has 16,000 members
- The /r/cpp subreddit has 185,000 members
- The /r/Cplusplus subreddit has 25,000 members
- CppCon is held annually and posts hundreds of talks
- Engine-specific forums are generally very active
- Many C++ GitHub repositories have 10,000+ stars and active Issues sections

2. Primitive Types and Literals

Types

Let's start with integers, which are surprising in two ways: how loosely defined they are and how many types there are. The type name itself is made up of one or more parts:

Part	Meaning
`signed`, `unsigned`, or none	If the type is signed or not. None means signed.
`short`, `long`, `long long`, or none	Size classification of the integer. Not an exact size! None means `int`.
`int` or none	Explicitly state that this is an integer. None states this implicitly.

Here's all 24 permutations, including the sizes in bits on common platforms:

C# Type	C++ Type	Windows Size	Unix Size
`short`	`short`	16	16
`short`	`short int`	16	16
`short`	`signed short`	16	16
`short`	`signed short int`	16	16
`ushort`	`unsigned short`	16	16
`ushort`	`unsigned short int`	16	16

C# Type	C++ Type	Windows Size	Unix Size
`int`	`int`	32	32
`int`	`signed`	32	32
`int`	`signed int`	32	32
`uint`	`unsigned`	32	32
`uint`	`unsigned int`	32	32
N/A	`long`	32	64
N/A	`long int`	32	64
N/A	`long int`	32	64
N/A	`signed long`	32	64
N/A	`signed long int`	32	64
N/A	`unsigned long`	32	64
N/A	`unsigned long int`	32	64
`long`	`long long`	64	64
`long`	`long long int`	64	64
`long`	`signed long long`	64	64
`long`	`signed long long int`	64	64
`ulong`	`unsigned long long`	64	64
`ulong`	`unsigned long long int`	64	64

There is also a type called `size_t` which is either a 32-bit or 64-bit unsigned integer, depending on the CPU being compiled for.

There are four 8-bit types:

C# Type	C++ Type	x86 and x64	ARM
`bool`	`bool`	N/A	N/A
`sbyte`	`char`	Signed	Unsigned
`sbyte`	`signed char`	Signed	Signed
`byte`	`unsigned char`	Signed	Signed

The types named with `char` are due to their original usage for characters in ASCII strings. There are also larger character types:

C# Type	C++ Type	Windows Size	Unix Size
N/A	`char8_t`	8	8
N/A	`char16_t`	16	16
N/A	`char32_t`	32	32
N/A	`wchar_t`	16	32

Next we have floating-point types, including a super high precision `long double` type:

C# Type	C++ Type	x86 Size	ARM Size
`float`	`float`	32	32
`double`	`double`	64	32
N/A	`long double`	80	128

There is no `decimal` type in C++, but libraries such as GMP provide similar functionality.

Given the uncertainty of size across CPU and OS, it's a best practice to avoid many of these types and instead use types that have specific sizes. These are found in the Standard Library or in game engine APIs. Here's how much simpler that makes everything:

Meaning	**C# Type**	**C++ Type**	**Unreal Type**
Boolean	`bool`	`bool`	`bool`
8-bit signed integer	`sbyte`	`int8_t`	`int8`
8-bit unsigned integer	`byte`	`uint8_t`	`uint8`
16-bit signed integer	`short`	`int16_t`	`int16`
16-bit unsigned integer	`ushort`	`uint16_t`	`uint16`
8-bit character	N/A	`char8_t`	`CHAR8`
16-bit character	`char`	`char16_t`	`CHAR16`
32-bit character	N/A	`char32_t`	`CHAR32`
32-bit signed integer	`int`	`int32_t`	`int32`
32-bit unsigned integer	`uint`	`uint32_t`	`uint32`
64-bit signed integer	`long`	`int64_t`	`int64`
64-bit unsigned integer	`ulong`	`uint64_t`	`uint64`
32-bit floating point number	`float`	`float`	`float`
128-bit floating point number	`decimal`	N/A	N/A

Literals

Now that we know all these types, let's express them by writing some literals. First, and most obviously, booleans:

Literal	Type	Value
`true`	`bool`	`1`
`false`	`bool`	`0`

Next are integers. They are written in four parts:

Part	Meaning
`0x`, `0X`, `0`, `0b`, `0B` or none	The chosen base: hexadecimal, octal, or binary. None means decimal.
`0123456789abcdefABCDEF'`, `01234567'`, or `01'`	Digits of the chosen base. `'` characters are optional separators like `_` in C#.
`u`, `U`, or none	If the integer is unsigned. None means signed for decimal and octal, unsigned for hexadecimal and binary.
`l`, `L`, `ll`, `LL`, or none	The size classification. None means "the smallest size that can fit the value" from the `int` size classification to `long` then to `long long`. Note: can be swapped with `u` or `U`, if specified

Here are some examples:

Literal	Type	Base	Signed	Size
`123`	`int`	Decimal (default)	Signed (default)	`int` (default)
`5000000000`	`long`	Decimal (default)	Signed (default)	`long` (default)
`123u`	`unsigned int`	Decimal (default)	Unsigned (explicit)	`int` (default)
`123ul`	`unsigned long`	Decimal (default)	Unsigned (explicit)	`long` (explicit)
`123lu`	`unsigned long`	Decimal (default)	Unsigned (explicit)	`long` (explicit)

Literal	Type	Base	Signed	Size
`0x123456`	`int`	Hexadecimal (explicit)	Signed (default)	`int` (default)
`0xffffffff`	`unsigned int`	Hexadecimal (explicit)	Unsigned (default)	`int` (default)
`0xfffffffff`	`long`	Hexadecimal (explicit)	Signed (default)	`long` (default)
`0xFFFFFFFFll`	`long long`	Hexadecimal (explicit)	Signed (default)	`long long` (explicit)
`0b10101010'01010101'10101010'01010101`	`unsigned int`	Binary (explicit)	Unsigned (default)	`int` (default)
`0123`	`int`	Octal (explicit)	Signed (default)	`int` (default)

Next up are floating point literals, which are also written in four parts parts:

Part	Meaning
`0x`, `0X`, or none	Choose hexadecimal, or none for decimal
`0123456789abcdefABCDEF.'`	Digits of the chosen base. `'` characters are optional separators like _ in C#. May end in `.` for whole numbers.
`e`, `e` then `+-` then `0123456789`, `p`, `p` then `+-` then `0123456789`, or none	Exponent `x` to multiply digits by `10^x`. Always required for hexadecimal and required for decimal if there's no `.` in the digits. `e` for decimal and `p` for hexadecimal.
`f`, `F`, `l`, `L`, or none	Size classification of `float` (`f`) or `long double` (`l`). None means `double`.

Here are some example floating point literals:

Literal	Type	Base
`12.34`	`double`	Decimal
`12.34f`	`float`	Decimal
`12.34F`	`float`	Decimal
`12.34e2`	`double`	Decimal
`12.34e-2`	`double`	Decimal

Literal	Type	Base
`12.34e-2f`	`float`	Decimal
`12.e1`	`double`	Decimal
`12'34.56'78f`	`float`	Decimal
`0x12p2`	`double`	Hexadecimal
`0x12.p2`	`double`	Hexadecimal
`0x12'34'56.78p2f`	`float`	Hexadecimal

Finally, we have character literals which take several forms:

Form	Meaning
`'c'`	`char` type if `c` fits, otherwise `int` type, with character `c`
`u8'c'`	`char8_t` type with UTF-8 character `c`
`u'c'`	`char16_t` type with UTF-16 character `c`
`U'c'`	`char32_t` type with UTF-32 character `c`
`L'c'`	`wchar_t` type with character `c`
`'abc'`	`int` type representing multiple characters `abc`

Characters can be anything in their set (e.g. UTF-8) except `'`, `\`, and the newline character. To get those, and other special characters, use an escape sequence:

Meaning	Escape Sequence	Note	Example
Single quote	`\'`		
Double quote	`\"`		
Question mark	`\?`		
Backslash	`\\`		
Bell	`\a`		
Backspace	`\b`		
Form feed	`\f`		
Line feed	`\n`		

Meaning	Escape Sequence	Note	Example
Carriage return	`\r`		
Tab	`\t`		
Vertical tab	`\v`		
Octal value	`\ABC`	`\ABC` is the octal value	`\0` is `NUL`
Hexadecimal value	`\xAB`	`\AB` is the hexadecimal value	`\x41` is `A`
16-bit Unicode code point	`\uABCD`	`\ABCD` is the code point	`\u03b1` is α
32-bit Unicode code point	`\UABCDEFGH`	`\ABCDEFGH` is the code point	`\U0001F389` is 🎉

Here are some example character literals:

Literal	Type	Decimal Value
`'A'`	`char`	`65`
`'?'`	`char`	`63`
`u8'A'`	`char8_t`	`65`
`u'α'`	`char16_t`	`945`
`U'\x1f389'`	`char32_t`	`127881`
`'ab'`	`int`	`127881`

Conclusion

C++ literals are similar to C# literals, but different in several ways. You can often write the exact same code in both languages and get the same effect. There are several edge cases though, so it's important to know some of these details about how the language works.

3. Variables and Initialization

Declaration

The basic form of a variable declaration should be very familiar to C# developers:

```
int x;
```

Just like in C#, we state the type of the variable, the variable's name, and end with a semicolon. We can also declare multiple variables in one statement:

```
int x, y, z;
```

Also like C#, these variables do not yet have a value. Consider trying to read the value of such a variable:

```
int x;
int y = x;
```

In C#, this would result in a compiler error on the second line. The compiler knows that `x` doesn't have a value, so it can't be read and assigned to `y`. In C++, this is known as "undefined behavior." When the compiler encounters undefined behavior, it is free to generate arbitrary code for the entire executable. It may or may not produce a warning or error to warn about this, meaning it may silently produce an executable that doesn't do what the author thinks it should do. It

is very important to never invoke undefined behavior and tools have been written to help avoid it.

This undefined behavior does have a purpose: speed. Consider this:

```
int localPlayerHealth;
foreach (Player p in players)
{
    if (p.IsLocal)
    {
        localPlayerHealth = p.Health;
        break;
    }
}
Debug.Log(localPlayerHealth);
```

We know that one player has to be the local player because that's how our game was designed, so it's safe to not initialize `localPlayerHealth` before the loop. Initializing it to `0` would be wasteful in this case, but the C# compiler doesn't know about our game design so it can't prove that we'll always find the local player and it forces us to initialize.

In C++, we're free to skip this initialization and assume the risk of undefined behavior if it turns out there really wasn't a local player in the `players` array. Alternatively, we can replicate the C# approach and just initialize the variable to be safe.

Initialization

C++ provides a lot of ways to initialize variables. We've already seen one above where a value is copied:

```
int x = y;
```

There are also some other ways that aren't in C#:

```
int x{}; // x is filled with zeroes, so x == 0
int x{123};
int x(123);
```

Many more types of initialization exist, but are specific to certain types such as arrays and classes. We'll cover these later in the book.

All of these initialization strategies can be combined when declaring multiple variables in one statement:

```
int a, b = 123, c{}, d{456}, e(789);
```

This results in these values:

Variable	Value
`a`	(Unknown)
`b`	`123`

Variable	Value
`c`	`0`
`d`	`456`
`e`	`789`

Type Deduction

In C#, we can use `var` to avoid needing to specify the type of our variables. Similarly, C++ has the `auto` keyword:

```
auto x = 123;
auto x{123};
auto x(123);
```

Also similar to C#, we can only use `auto` when there is an initializer. The following isn't allowed:

```
auto x;
auto x{};
```

It's important to remember that `x` is just as strongly-typed as if `int` were explicitly specified. All that's happening here is that the compiler is figuring out what type the variable should be rather than us typing it out manually.

An alternative approach, much less frequently seen, is to use the `decltype` operator. This resolves to the type of its parameter:

```
int x;
decltype(x) y = 123; // y is an int
```

Lastly, since C++17 the `register` keyword has been deprecated:

```
register int x = 123;
```

It used to request that the variable be placed into a CPU register rather than in RAM, such as on the stack. Compilers have long ignored this request, so it’s best to avoid this keyword now.

Identifiers

The rules for naming C++ identifiers are similar to the rules for C#. They must begin with a letter, underscore, or any non-digit Unicode character. After that, they can contain any Unicode character except some really strange ones.

Additionally, there are some restrictions on the names we can choose:

Restriction	**Example**	**Where**
All keywords	`int for`	All code
`operator` then an operator symbol	`operator+`	All code
`~` then a class name	`~MyClass`	All code
Any name beginning with double underscores	`int __x`	All code except the Standard Library
Any name beginning with an underscore then a capital letter	`int _X`	All code except the Standard Library
Any name beginning with an underscore	`int _x`	All code in the global namespace except the Standard Library

There is no equivalent to C#'s "verbatim identifiers" (e.g. `int @for`) to work around the keyword restriction.

Pointers

Like C#, at least when “unsafe” features are enabled, C++ has pointer types. The syntax is even similar:

```
int* x;
int * x;
int *x;
```

The placement of the `*` is flexible, just like in C#. However, declaring multiple variables in one line is different in C++. Consider this declaration:

```
int* x, y, z;
```

The type of `y` differs between the languages since the `*` only attaches to one variable in C++:

Language	Type of `x`	Type of `y`	Type of `z`
C#	`int*`	`int*`	`int*`
C++	`int*`	`int`	`int`

To make all three variables into pointers in C++, add a `*` to each:

```
int *x, *y, *z;
```

Or omit a `*` so that only some are pointers:

```
int *x, y, *z; // x and z are int*, y is int
```

We'll cover how to actually use pointers more in depth later in the book.

References

C++ has two kinds of references: “lvalue” and “rvalue.” Just like with pointers, these are an annotation on another type:

```
// lvalue references
int& x;
int & x;
int &x;

// rvalue references
int&& x;
int && x;
int &&x;
```

When declaring more than one variable per statement, the same rule applies here: `&` or `&&` only attaches to one variable:

```
int &x, y; // x is an int&, y is an int
```

Taken all together, this means we can declare several variables per statement and each can have their own modifier on the stated type:

```
int a, *b, &c, &&d;
```

The variables get these types:

Variable	Type
a	int
b	int*
c	int&
d	int&&

We'll dive into the details of how lvalue references and rvalue references work later in the book. For now, it's important to know that they are like non-nullable pointers. This means we *must* initialize them when they are declared. All of the above lines will fail to compile since we didn't. So let's correct that:

```
int x = 123;
int& y = x;

int&& z = 456;
```

Here we have `y` as an "lvalue reference to an `int`." We initialize it to an lvalue, which is essentially anything with a name. `x` has a name and is the right type: `int`. The result is that `y` now references `x`.

`z` is an "rvalue reference to an `int`." An rvalue is essentially anything without a name. We initialize it to `456` which has no name but does have the right type: `int`. This means that `z` now references `456`.

Putting this back together, we end up with multiple variables being declared and initialized when required like this:

```
int x = 123;
int a, *b, &c = x, &&d = 456;
```

Conclusion

At a high level, variables in C++ are similar to C#. In the details though, there are very important differences. The undefined behavior stemming from not initializing them, pointer and reference characters only applying to one variable in multiple declaration, various new kinds of initialization syntax, and the presence of both lvalue and rvalue references all make for a pretty different landscape even in this basic category of variables.

4. Functions

Declaration and Definition

Functions in C++ can be split into two parts. The first is the function declaration, which states its signature without stating how it works. To do this, simply add a semicolon after the signature:

```
int Add(int a, int b);
```

Now let's write the second part: the function's definition. This also contains the function's signature but includes the body too:

```
int Add(int a, int b)
{
    return a + b;
}
```

So why does C++ have both a declaration and a definition when C# only has the definition? It mostly has to do with how C++ is compiled. We'll cover that more in depth in the next chapter, but for now it's important to know that C++ is compiled from top to bottom in a source file. A function definition or declaration makes it available to be referenced by code further down in the file. Here's an approximation of what the compiler does:

```
// Start compiling here and read down the file's lines...

// There is no function `Add` at this point
```

```
// It doesn't matter that there is an `Add` later on
// This is a compile error
int four = Add(1, 3);

// Declaration of the `Add` function
// `Add` can now be referenced
int Add(int a, int b);

// Refers to `Add`, which the compiler knows about
// It doesn't matter that there's no definition yet
// The compiler trusts that the definition will come
later
// If it doesn't, there will be an error
int three = Add(2, 1);

// Definition of the `Add` function
// The programmer has fulfilled the promise to define it
// The compiler now knows what to do when `Add` is called
int Add(int a, int b)
{
    return a + b;
}
```

This is also OK:

```
// Definition of the `Add` function
// `Add` can now be referenced
// Also says what to do when `Add` is called
```

```
int Add(int a, int b)
{
    return a + b;
}

// Refers to `Add`, which the compiler knows about
int three = Add(2, 1);
```

So we can use a function declaration to reorder our code, even though it's compiled from top to bottom. We can just state the signature at the top and leave the body until later on. In the next chapter, we'll talk about header files and this will become much more important. For now, it's good to know that C++ functions are commonly seen with and without a declaration.

One last quirk: it's possible to declare more than one function in a statement just like `int a, b;` declares two variables. This is very rarely seen and should generally be avoided in favor of the single-declaration form.

```
// Two functions:
//   int Add(int a, int b)
//   int Sub(int a, int b)
int Add(int a, int b), Sub(int a, int b);
```

As with variables, both functions share the same return type.

Optional Parameter Names and Void

There's a strange aspect of parameter names in C++: they're optional! This declaration is totally fine:

```
int Add(int, int);
```

After all, we're just telling the compiler the signature of the function and the parameter names are irrelevant to that. Perhaps more strangely, we can omit the parameter names from function definitions!

```
// A very poor implementation of addition...
int Add(int a, int)
{
    return a + 1;
}
```

This is sometimes useful when a certain function signature is required but the parameters aren't actually used in the body of the function. Consider a function meant to be used as an event handler:

```
void OnPlayerSpawned(Vector3)
{
    NumSpawns++;
}
```

This function doesn't care where the player spawned because all it's doing is keeping track of a statistic. So we can omit the parameter name for a couple reasons. First, it tells the reader that this parameter isn't important in the function so it isn't even given a name that needs to be memorized. Second, it tells the compiler not to complain about an unused variable. After all, we can't use a varaible without a name in the first place. Sometimes we see a middle ground in C++ code where the name is stated inside a comment to gain only the second benefit but not the first:

```
void OnPlayerSpawned(Vector3 /* position */)
{
    NumSpawns++;
}
```

If the function takes no parameters at all, it may optionally state this explicitly by putting `void` where the parameters would normally go:

```
uint64_t GetCurrentTime(void);
```

Whether to add `void` or not is purely a stylistic choice.

Automatic Return Types

C++ variables can be declared with an `auto` type, similar to `var` in C#. In C++, function return types can also be declared as `auto`:

```
auto Add(int a, int b)
{
    return a + b;
}
```

Just like with variables, the compiler figures out what the return type should be. In this case, it's just `int` since that's what we get when adding two `int` values together.

We can also specify the return type *after* the parameter list if we put `auto` before the function name:

```
auto Add(int a, int b) -> int
{
    return a + b;
}
```

In this case, we're explicitly stating the return type. It is not automatically determined by the compiler, even though we have to still add `auto` before the function name. This alternative syntax is sometimes useful when the return type is very complex. We'll see some examples later in the book when we tackle function pointers and templates.

Default Arguments

As in C#, default arguments are allowed as long as there aren't any non-defaulted parameters after the first one. It's a little different in C++ though due to split between declaration and definition. If the function has both, the default arguments are specified in the declaration:

```
// Function declaration states the default argument
values
void SpawnPlayer(Vector3 position, float speed=0.0f);

// Function definition omits them
void SpawnPlayer(Vector3 position, float speed)
{
    // ...
}
```

If there's no declaration, then the default arguments are added to the definition:

```
void SpawnPlayer(Vector3 position, float speed=0.0f)
{
    // ...
}
```

Variadic Functions

As in C#, functions may take a variable number of parameters. This works *really* differently in C++ though. It's not deprecated like `register` variables are, but it's often considered a bad practice to even use the feature. Still, let's see how they look:

```
// The `...` comes after all the normal parameters
// It means "0 or more parameters of any types go here"
void PrintLog(LogLevel level, ...)
{
    // ...
}
```

The function should then call `va_start`, `va_arg`, and `va_end` in order to get the arguments. This is quite type-unsafe and a very clunky interface, which is part of why the feature should generally not be used. There are several alternatives that are preferred instead, but many are more advanced features that will be discussed later on in the book. For now, let's discuss a simple one: overloading.

Overloading

As in C#, functions may be overloaded in the sense that more than one function may have the same name. When the function is called, the compiler figures out which of these identically-named functions should actually be called.

```
// Get the player's score given their ID
int GetPlayerScore(int playerId);

// Get the local player's score
int GetPlayerScore();

// Get the score of the player at a given position
int GetPlayerScore(Vector3 position);
```

These functions vary by the type of argument and the number of arguments. Now we can write code like this:

```
score = GetPlayerScore(myPlayerId);
score = GetPlayerScore();
score = GetPlayerScore(myPosition);
```

In this case, the compiler will generate calls to the three functions we declared in the same order.

Ref, Out, and In parameters

In C#, parameters can be declared with the `ref`, `in`, and `out` keywords. Each of these change the parameter to be a pointer to the passed value. In C++, these keywords don't exist. Instead, we use some conventions:

```
// Alternative to `ref`
// Use an lvalue reference, which is like a non-nullable
pointer
void MovePlayer(Player& player, Vector3 offset)
{
    player.position += offset;
}

// Alternative to `in`
// Use a constant lvalue reference
// `const` means it can't be changed
void PrintPlayerName(const Player& player)
{
    DebugLog(player.name);
}

// Alternative to `out`
// Just use return values
ReallyBigMatrix ComputeMatrix()
{
    ReallyBigMatrix matrix;
```

```
    // ...math goes here...
    return matrix
}

// Another alternative to `out`
// Use lvalue reference parameters
void ComputeMatrix(ReallyBigMatrix& mat1,
ReallyBigMatrix& mat2)
{
    mat1 = /* math for mat1 */;
    mat2 = /* math for mat2 */;
}

// Another alternative to `out`
// Pack the outputs into a return value
tuple<ReallyBigMatrix, ReallyBigMatrix> ComputeMatrix()
{
    return make_tuple(/* math for mat1 */, /* math for
mat2 */);
}
```

In the case of `ref`, C++ functions typically use an lvalue reference. If the reference needs to be nullable, which isn't allowed for a C# `ref` parameter, a pointer (`Player*`) can be used instead.

For `in`, a const lvalue reference is typically used. We haven't covered `const` yet, but suffice to say it doesn't allow changes to the variable. Writing `player.score = 0;` would cause a compiler error. This is broadly similar to what would happen with `out` parameters in

C#. Again, a pointer (`const Player* player`) can be used if the parameter needs to sometimes be null.

`out` parameters are usually written by just returning them. In case more than one return value is needed, there are a couple of main options. First, lvalue reference parameters can be taken. This has the unfortunate downside that they can be read from before being assigned to so they might be inadvertently used as input. It's also unclear to callers whether they're providing arguments for input, output, or both. Second is the much-preferred option: pack all the outputs into a `struct` and return it. We haven't talked about structs or templates, which are similar to C# generics, but the Standard Library's `tuple` type and `make_tuple` helper function are shown here as roughly the alternatives to C# tuples.

Static Variables

Local variables within functions may be declared `static`. Similar to `static` fields of classes and structs in C#, this means that the variable has only one instance. A `static` C++ local variable has only one instance across all calls to the function:

```
int GetNextId()
{
    static int id = 0;
    id++;
    return id;
}

GetNextId(); // 1
GetNextId(); // 2
GetNextId(); // 3
```

In this example we have an `id` local variable that is `static`. There will be only one `id` across all calls to `GetNextId`. It's like `id` is a global variable, but it can only be referred to within the function where it's declared. This can be very convienient, but also be very surprising, just like `static` fields in C#.

Constexpr

Finally for this chapter, functions may be marked with `constexpr`. This means that the function can be evaluated at compile time. For example:

```
constexpr int GetSquareOfSumUpTo(int n)
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
    {
        sum += i;
    }
    return sum * sum;
}
```

This function can then be evaluated at compile time in order to generate a constant:

```
DebugLog(GetSquareOfSumUpTo(5000));

// equivalent to...

DebugLog(1020530960);
```

The function can also be evaluated at runtime, such as when its parameters are dependent on runtime values:

```
int n = file.ReadInt();
DebugLog(GetSquareOfSumUpTo(n));
```

This means that normal C++ can be reused for both compile time and runtime work. There's usually no need to run scripts in another language in order to generate C++ files. The types and functionality the program is already made up of are usable at compile time with this mechanism.

There are some restrictions to what's possible in a `constexpr` function though. Since they were introduced in C++11, each version has relaxed these restrictions. Still, some features like `static` local variables and `goto` aren't allowed even in C++20.

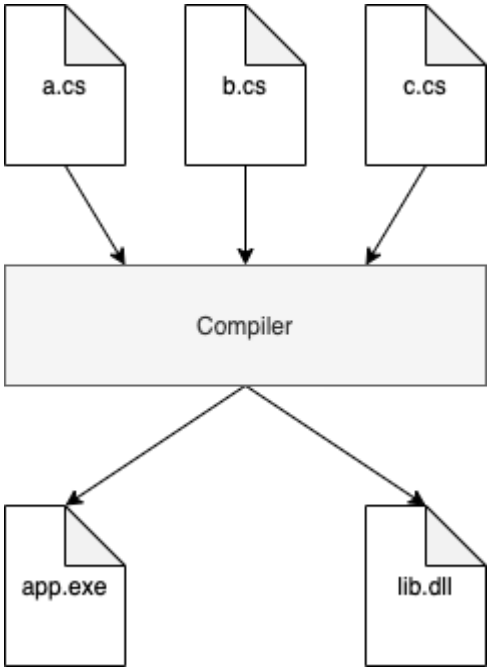
Conclusion

Broadly speaking, functions are very similar between C# and C++. There are *many* differences though. These differences span syntactic quirks like how `ref` parameters are declared all the way to radically different features like compile-time function execution and static local variables. As we progress through the book, we'll learn about many more types of functions including member functions ("methods") and lambdas!

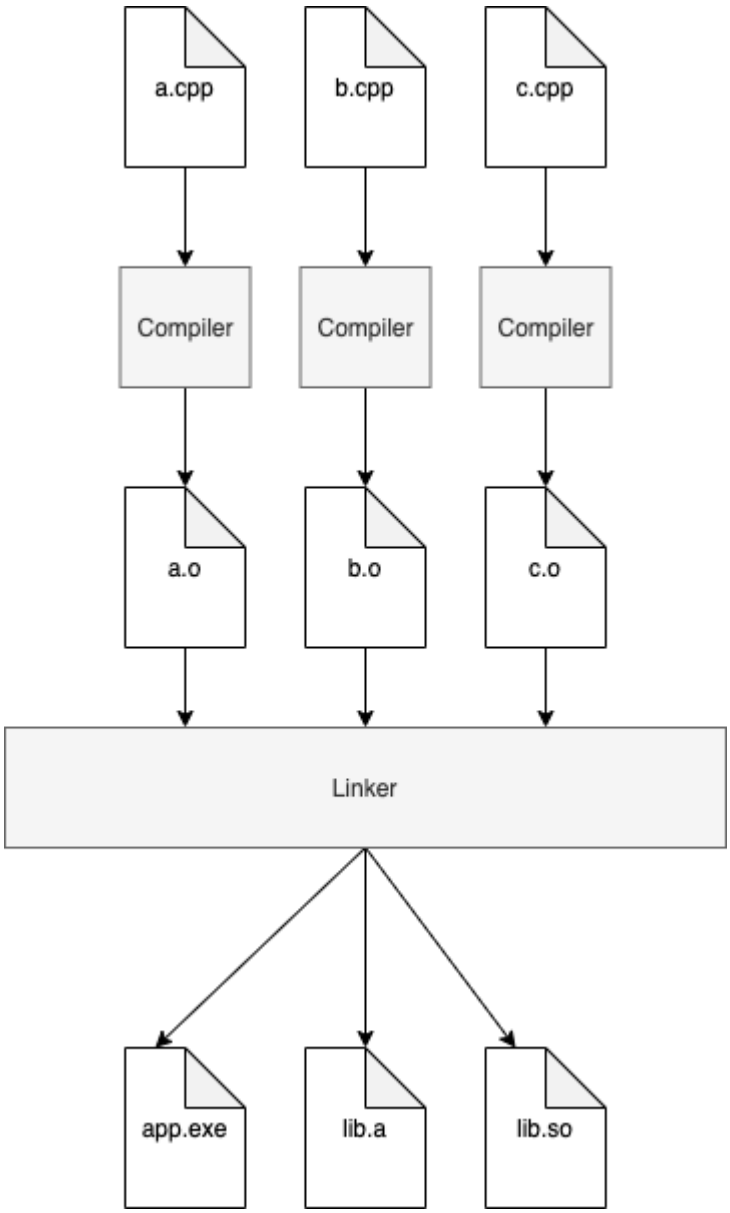
5. Build Model

Compiling and Linking

With C#, we compile all our source code files (`.cs`) into an assembly such as an executable (`.exe`) or a library (`.dll`).



With C++, we compile all our translation units (source code files with `.cpp`, `.cxx`, `.cc`, `.C`, or `.c++`) into object files (`.obj` or `.o`) and then link them together into an executable (`app.exe` or `app`), static library (`.lib` or `.a`), or dynamic library (`.dll` or `.so`).

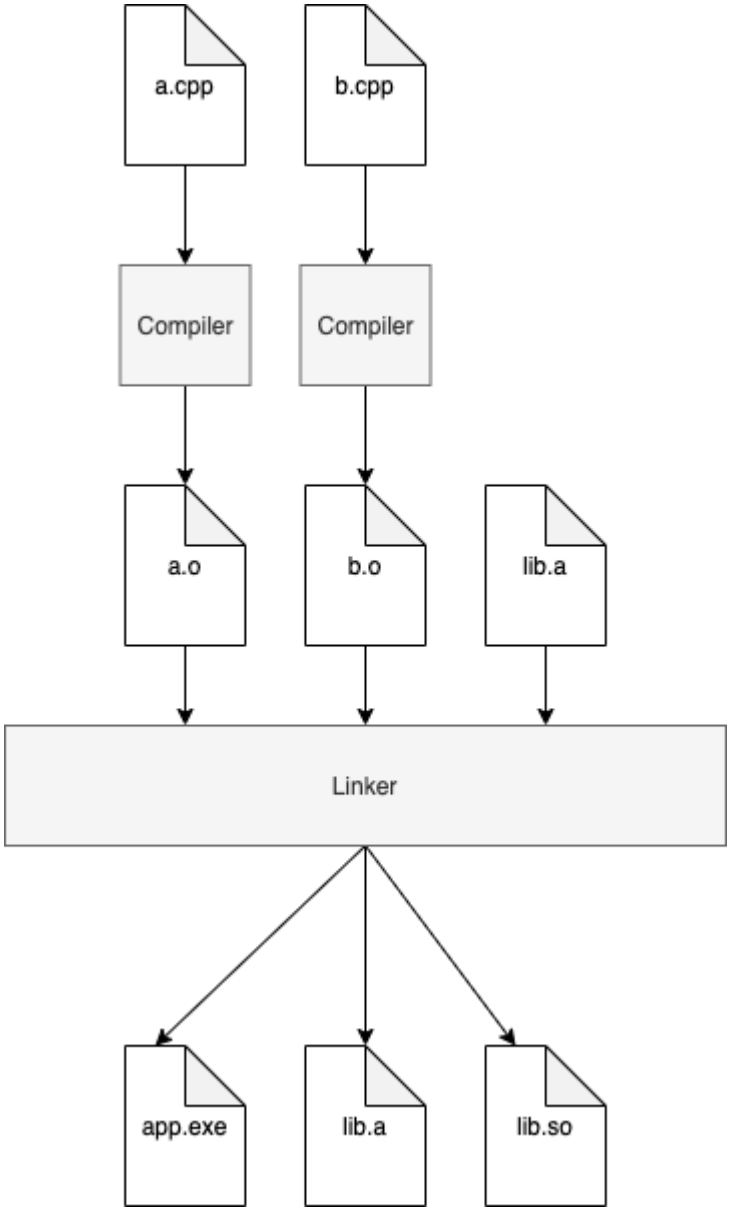


If any of the source code files changed, we recompile them to generate a new object files and then run the linker with all the unchanged object files too.

This model brings up a couple of questions. First, what is an object file? This is known as an “intermediate” file since it’s neither the source code nor an output file like an executable. The C++ language standard doesn’t say anything about what the format of this file is. In practice, it’s a binary file that is specific to a particular version of a

particular compiler configured with particular settings. If the compiler, version, or settings change, all the code needs to be rebuilt.

Second, what is the difference between a static library and a dynamic library? A dynamic library is very similar to a dynamic library in C#. It's a library of machine code, just like an executable. However, it can be loaded and unloaded by an executable or other dynamic library at runtime. A static library, on the other hand, can only be loaded at *compile time* and can never be unloaded. In this way, it functions more like just another object file:



Because static libraries are available at build time, the linker builds them directly into the resulting executable. This means there's no need to distribute a separate dynamic library file to end users, no need to open it from the file system separately, and no possibily of overriding its location such as by setting the `LD_LIBRARY_PATH` environment variable.

Critically for performance, all calls into functions in the static library are just normal function calls. This means there's no indirection through a pointer that is set at runtime when a dynamic library is loaded. It also means that the linker can perform "link time optimizations" such as inlining these functions.

The main downsides stem from needing the static libraries to be present at compile time. This makes them unsuitable for tasks such as loading user-created plugins. Perhaps most importantly for large projects, they must be linked in every build even if just one small source file was changed. Link times grow proportionally and can hinder rapid iteration. As a result, sometimes dynamic libraries will be used in development builds and static libraries will be used in release builds.

We won't discuss the specifics of how to run the compiler and linker in this book. This is heavily dependent on the specific compiler, OS, and game engine being used. Usually game engines or console vendors will provide documentation for this. Also typical is to use an IDE like Microsoft Visual Studio or Xcode that provides a "project" abstraction for managing source code files, compiler settings, and so forth.

Header Files and the Preprocessor

In C#, we add `using` directives to reference code in other files. C++ has a similar "module" system added in C++20 which we'll cover in a future chapter in this book. For now, we'll pretend like that doesn't exist and only discuss the way that C++ has traditionally been built.

Header files (`.h`, `.hpp`, `.hxx`, `.hh`, `.H`, `.h++`, or no extension) are by far the most common way for code in one file to reference code in another file. These are simply C++ source code files that are intended to be copy-and-pasted into another C++ source code file. The copy-and-paste operation is performed by the preprocessor.

Just like in C#, preprocessor directives like `#if` are evaluated before the main phase of compilation. There is no separate preprocessor executable that must be called to produce an intermediate file that the compiler receives. Preprocessing is simply an earlier step for the compiler.

C++ uses a preprocessor directive called `#include` to copy and paste a header file's contents into another header file (`.h`) or a translation unit (`.cpp`). Here's how it looks:

```
// math.h
int Add(int a, int b);


// math.cpp
#include "math.h"
int Add(int a, int b)
{
```

```
    return a + b;
}
```

The `#include "math.h"` tells the preprocessor to search the directory that `math.cpp` is in for a file named `math.h`. If it finds such a file, it reads its contents and replaces the `#include` directive with them. Otherwise, it searches the "include paths" it's been configured with. The C++ Standard Library is implicitly searched. If `math.h` isn't found in any of these locations, the compiler produces an error.

Afterward, `math.cpp` looks like this:

```
int Add(int a, int b);
int Add(int a, int b)
{
    return a + b;
}
```

Recall from the previous chapter that the first `Add` is a function *declaration* and the second is a function *definition*. Since the signatures match, the compiler knows we're defining the earlier declaration.

So far we've split the declaration and definition across two files, but without much benefit. Now let's make this pay off by adding another translation unit:

```
// user.cpp
#include "math.h"
int AddThree(int a, int b, int c)
```

```
{
    return Add(a, Add(b, c));
}
```

This shows how `user.cpp` can add the same `#include "math.h"` to access the declaration of `Add`, resulting in this:

```
int Add(int a, int b);
int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
```

Now the compiler will encounter the declaration of `Add` and be OK with `AddThree` calling it even though there's no definition of `Add` yet. It simply makes a note in the object file it outputs (`user.obj`) that `Add` is an unsatisfied dependency.

When the linker executes, it reads in `user.obj` and `math.obj`. `math.obj` contains the definition of `Add` and `user.obj` contains the definition of `AddThree`. At that point, the linker really needs the definition of `Add`, so it uses the one it found in `math.obj`.

There is an alternative version of `#include` that's commonly seen:

```
#include <math.h>
```

This version is meant to search just for the C++ Standard Library and other header files that the compiler provides. For example, Microsoft Visual Studio allows `#include <windows.h>` to make

Windows OS calls. This is useful to disambiguate file names that are both in the application's codebase and provided by the compiler. Imagine this program:

```
#include "math.h"
bool IsNearlyZero(float val)
{
    return fabsf(val) < 0.000001f;
}
```

`fabsf` is a function in the C Standard Library to take the absolute value of a `float`. When the preprocessor runs with the quotes version of `#include` it finds our `math.h`, so we get this:

```
int Add(int a, int b);

bool IsNearlyZero(float val)
{
    return fabsf(val) < 0.000001f;
}
```

Then the compiler can't find `fabsf` so it errors. Instead, we should use the angle brackets version of `#include` since we're looking for the compiler-provided `math.h`:

```
#include <math.h>
bool IsNearlyZero(float val)
{
```

```
    return fabsf(val) < 0.000001f;
}
```

This produces what we wanted:

```
float fabsf(float arg);
// ...and many, many more math function declarations...

bool IsNearlyZero(float val)
{
    return fabsf(val) < 0.000001f;
}
```

Also note that we can specify paths in the `#include` that correspond to a directory structure:

```
#include "utils/math.h"
#include <nlohmann/json.hpp>
```

Finally, while it's esoteric and usually best avoided, there is nothing stopping us from using `#include` to pull in non-header files. We can `#include` any file as long as the result is legal C++. Sometimes `#include` is even placed in the middle of a function to fill in part of its body!

ODR and Include Guards

C++ has what it calls the “one definition rule,” commonly abbreviated to ODR. This says that there may be only one definition of something in a translation unit. This includes variables and functions, which presents us some problems as our codebase grows. Imagine we’ve expanded our math library and added a vector math library on top of it:

```
// math.h
int Add(int a, int b);
float PI = 3.14f;


// vector.h
#include "math.h"
float Dot(float aX, float aY, float bX, float bY);


// user.cpp
#include "math.h"
#include "vector.h"
int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
```

```
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

Here we have `vector.h` using `#include` to pull in `math.h`. We also have `user.cpp` using `#include` to pull in both `vector.h` and `math.h`. This is a good practice since it avoids an implicit dependency on `math.h` that would break if `vector.h` was ever changed to remove the `#include "math.h"`. Still, we're about to see that this presents a problem. Let's look at `user.cpp` after the preprocessor has replaced the `#include "math.h"` directive:

```
int Add(int a, int b);
float PI = 3.14f;
#include "vector.h"
int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

Now the compiler replaces the `#include "vector.h"`:

```
int Add(int a, int b);
float PI = 3.14f;
#include "math.h"
```

```
float Dot(float aX, float aY, float bX, float bY);
int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

Finally, it replaces the `#include "math.h"` from the contents of `vector.h` that it copied in:

```
int Add(int a, int b);
float PI = 3.14f;
int Add(int a, int b);
float PI = 3.14f;
float Dot(float aX, float aY, float bX, float bY);
int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

Multiple *declarations* of the `Add` function are OK because they're not *definitions* so they don't violate the ODR. The compiler simply ignores the duplicate declarations.

The *definition* of `PI`, on the other hand, is most certainly a definition. Having two definitions of the same variable name violates the ODR and we get a compiler error.

To work around this, we add what's called an "include guard" to our header files. There are two basic forms this can take, but both make use of the preprocessor. Here's the first form in `math.h`:

```
#if (!defined MATH_H)
#define MATH_H

int Add(int a, int b);
float PI = 3.14f;

#endif
```

This makes use of the `#if`, `#define`, and `#endif` directives, which are similar to their C# counterparts. The only real difference in this case is the use of `!defined MATH_H` in C++ instead of just `!MATH_H` in C#.

One variant of this is to make use of a C++-only `#ifndef MATH_H` as a sort of shorthand for `#if (!defined MATH_H)`:

```
#ifndef MATH_H
#define MATH_H

int Add(int a, int b);
```

```
float PI = 3.14f;

#endif
```

In either case, we choose a naming convention and apply our file name to it to generate a unique identifier for the file. There are *many* popular forms for this including these:

```
math_h
MATH_H
MATH_H_
MYGAME_MATH_H
```

To avoid needing to come up with unique names, all common compilers offer the non-standard `#pragma once` directive:

```
#pragma once

int Add(int a, int b);
float PI = 3.14f;
```

Regardless of the form chosen, let's look at how this helps avoid the ODR violation. Here's how `user.cpp` looks after all the `#include` directives are resolved: (indentation added for clarity)

```
#ifndef MATH_H
#define MATH_H
```

```
    int Add(int a, int b);
    float PI = 3.14f;

#endif

#ifndef VECTOR_H
#define VECTOR_H

    #ifndef MATH_H
    #define MATH_H

        int Add(int a, int b);
        float PI = 3.14f;

    #endif

    float Dot(float aX, float aY, float bX, float bY);

#endif

int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

On the first line (`#ifndef MATH_H`), the preprocessor finds that `MATH_H` isn't defined so it keeps all the code until the `#endif`. That includes a `#define MATH_H`, so now it's defined.

Likewise, the `#ifndef VECTOR_H` succeeds and allows `VECTOR_H` to be defined. The nested `#ifndef MATH_H`, however, fails because `MATH_H` is now defined. Everything until the matching `#endif` is stripped out.

In the end, we have this result:

```
int Add(int a, int b);
float PI = 3.14f;

float Dot(float aX, float aY, float bX, float bY);

int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}
```

The duplicate definition of `PI` has been effectively removed from the translation unit by the include guard, so we no longer get a compiler error for the ODR violation.

Inline

Even with the ODR compiler error fixed, we still have a problem: a linker error. The reason for this is that the `vector.cpp` translation unit also contains a copy of `PI`. Here's how it looks originally:

```
#include "vector.h"

float Dot(float aX, float aY, float bX, float bY)
{
    return Add(aX*bX, aY+bY);
}
```

Here it is after the preprocessor resolves the `#include` directives:

```
#ifndef VECTOR_H
#define VECTOR_H

    #ifndef MATH_H
    #define MATH_H

        int Add(int a, int b);
        float PI = 3.14f;

    #endif

    float Dot(float aX, float aY, float bX, float bY);
```

```
#endif

float Dot(float aX, float aY, float bX, float bY)
{
    return Add(aX*bX, aY+bY);
}
```

Remember that each translation unit is compiled separately. In *this translation unit*, `MATH_H` and `VECTOR_H` have not been set with `#define` as they were in the `user.cpp` translation unit. So both of the include guards succeed and we get this:

```
int Add(int a, int b);
float PI = 3.14f;

float Dot(float aX, float aY, float bX, float bY);

float Dot(float aX, float aY, float bX, float bY)
{
    return Add(aX*bX, aY+bY);
}
```

That's great for the purposes of compiling this translation unit since there are no duplicate definitions to violate the ODR. Compilation will succeed, but linking will fail.

The reason for the linker error is that, by default, we can't have duplicate definitions of `PI` at link time either. If we want to do that, we need to add the `inline` keyword to `PI` to tell the compiler that

multiple definitions should be allowed. That'll result in these translation units:

```
// user.cpp
int Add(int a, int b);
inline float PI = 3.14f;

float Dot(float aX, float aY, float bX, float bY);

int AddThree(int a, int b, int c)
{
    return Add(a, Add(b, c));
}
bool IsOrthogonal(float aX, float aY, float bX, float bY)
{
    return Dot(aX, aY, bX, bY) == 0.0f;
}


// vector.cpp
int Add(int a, int b);
inline float PI = 3.14f;

float Dot(float aX, float aY, float bX, float bY);

float Dot(float aX, float aY, float bX, float bY)
{
```

```
    return Add(aX*bX, aY+bY);
}
```

It may seem strange that `inline` is a keyword applied to variables. The historical reason for this is that it was originally a hint to the compiler that it should inline *functions* but, like the `register` keyword, this was non-binding and virtually always ignored. It's come to mean "multiple definitions are allowed" instead, so it can now be applied to both variables and functions.

For example, we could add a function *definition* to `math.h` as long as it's `inline`:

```
inline int Sub(int a, int b)
{
    return a - b;
}
```

This is often avoided though because any change to the function will require recompiling all of the translation units that include it, directly or indirectly, which may take quite a while in a big codebase.

Linkage

Finally for this chapter, C++ has the concept of "linkage." By default, variables like `PI` have external linkage. This means it can be referenced by other translation units. For example, say we added a variable to `math.cpp`:

```
float SQRT2 = 1.4f;
```

Now say we want to reference it from `user.cpp`. The `#include "math.h"` won't work because `SQRT2` is in `math.cpp`, not `math.h`. We can still reference it using the `extern` keyword:

```
extern float SQRT2;

float GetDiagonalOfSquare(float widthOrHeight)
{
    return SQRT2 * widthOrHeight;
}
```

This is similar to a function declaration in that we're telling the compiler to trust us and pretend a `float` exists with the name `SQRT2`. So when it compiles `user.cpp` it makes a note in the `user.obj` object file that we haven't yet satisfied the dependency for `SQRT2`. When the compiler compiles `math.cpp`, it makes a note that there is a `float` named `SQRT2` available for linking.

Later on, the linker runs and reads in `user.obj` as well as all the other object files including `math.obj`. While processing `user.obj`, it reads that note from the compiler saying that the definition of `SQRT2`

is missing and it goes looking through the other object files to find it. Lo and behold, it finds a note in `math.obj` saying that there's a `float` named `SQRT2` so the linker makes `GetDiagonalOfSquare` refer to that variable.

Quick note: the `extern` keyword can also be applied in `math.cpp`, but this has no effect since external linkage is the default. Still, here's how it'd look:

```
extern float SQRT2 = 1.4f;
```

One way to prevent this behavior is to add the `static` keyword to `SQRT2`. This changes the linkage to "internal" and prevents the compiler from adding that note to `math.obj` to say that a `float` variable named `SQRT2` is available for linking.

```
static float SQRT2 = 1.4f;
```

Now if we try to link `user.obj` and `math.obj`, the linker can't find any available definition of `SQRT2` in any of the object files so it produces an error.

Both `extern` and `static` can be used with functions, too. For example:

```
// math.cpp
int Sub(int a, int b)
{
    return a - b;
}
```

```cpp
static int Mul(int a, int b)
{
    return a * b;
}


// user.cpp
extern int Sub(int a, int b);

int SubThree(int a, int b, int c)
{
    return Sub(Sub(a, b), c);
}

extern int Mul(int a, int b); // compiler error: Mul is
`static`
```

Conclusion

In this chapter we've seen C++'s very different approach to building source code. The "compile then link" approach combined with header files has domino effects into the ODR, linkage, and include guards. We'll go into C++20's module system that solves a lot of these problems and results in a much more C#-like build model later on in the book, but header files will still be very relevant even with modules. There's also a lot more detail to go into with respect to the ODR and linkage, but we'll cover that incrementally as we introduce more language concepts like templates and thread-local variables.

6. Control Flow

If and Else

Let's start with the lowly `if` statement, which is just like in C#:

```
if (someBool)
{
    // ... execute this if someBool is true
}
```

Unlike C#, there's an optional initialization statement that can be tacked on to the beginning. It's like the first statement of a `for` loop and is usually used to declare a variable scoped to the `if` statement. Here's how it's typically used:

```
if (ResultCode code = DoSomethingThatCouldFail(); code ==
FAILURE)
{
    // ... execute this if DoSomethingThatCouldFail
returned FAILURE
}
```

The `else` part is just like `C#`:

```
if (someBool)
{
    // ... execute this if someBool is true
```

```
}
else
{
    // ... execute this if someBool is false
}
```

Goto and Labels

The `goto` statement is also similar to in C#. We create a label and then name it in our `goto` statement:

```
void DoLotsOfThingsThatMightFail()
{
    if (!DoThingA())
    {
        goto handleFailure;
    }
    if (!DoThingB())
    {
        goto handleFailure;
    }
    if (!DoThingC())
    {
        goto handleFailure;
    }

    handleFailure:
        DebugLog("Critical operation failed. Aborting
program.");
        exit(1);
}
```

Like in C#, the label to `goto` must be in the same function. Unlike in C#, the label can't be inside of a `try` or `catch` block.

One subtle difference is that a C++ `goto` can be used to skip past the declaration of variables, but not the initialization of them. For example:

```
void Bad()
{
    goto myLabel;
    int x = 1; // Un-skippable initialization
    myLabel:
    DebugLog(x);
}

void Ok()
{
    goto myLabel;
    int x; // No initialization. Can be skipped.
    myLabel:
    DebugLog(x); // Using uninitialized variable
}
```

As with any use of an uninitialized variable, this is undefined behavior and will likely lead to severe errors. Care should be taken to ensure that the variable is eventually initialized before it's read.

Switch

C++ `switch`, `case`, and `default` are similar to their C# counterparts:

```
switch (someVal)
{
    case 1:
        DebugLog("someVal is one");
        break;
    case 2:
        DebugLog("someVal is two");
        break;
    case 3:
        DebugLog("someVal is three");
        break;
    default:
        DebugLog("Unhandled value");
        break;
}
```

One difference is that a `case` that's not empty can omit the `break` and "fall through" to the next `case`. This is sometimes considered error-prone, but can also reduce duplication. These two are equivalent:

```
// C#
switch (someVal)
{
    case 3:
```

```
        DoAtLeast3();
        DoAtLeast2();
        DoAtLeast1();
        break;
    case 2:
        DoAtLeast2();
        DoAtLeast1();
        break;
    case 1:
        DoAtLeast1();
        break;
}

// C++
switch (someVal)
{
    case 3:
        DoAtLeast3();
    case 2:
        DoAtLeast2();
    case 1:
        DoAtLeast1();
}
```

Another difference that curly braces are required in a `case` in order to declare variables:

```
switch (someVal)
{
    case 1:
    {
        int points = CalculatePoints();
        DebugLog(points);
        break;
    }
    case 2:
        DebugLog("someVal is two");
        break;
    case 3:
        DebugLog("someVal is three");
        break;
}
```

C++ `switch` statements also support initialization statements, much like with `if`:

```
switch (ResultCode code = DoSomethingThatCouldFail();
code)
{
    case FAILURE:
        DebugLog("Failed");
        break;
    case SUCCESS:
        DebugLog("Succeeded");
```

```
        break;
    default:
        DebugLog("Unhandled error code");
        DebugLog(code);
        break;
}
```

Unlike C#, a `switch` can only be used on integer and enumeration types. An chain of `if` and `else` is needed to handle anything else:

```
if (player == localPlayer)
{
    // .. handle the local player
}
else if (player == adminPlayer)
{
    // .. handle the admin player
}
```

C#'s pattern matching is also not supported, so there's no ability to write `case int x:` to match all `int` values and bind their value to `x`. There's also no `when` clauses, so we can't write `case Player p when p.NumLives > 0:`. Instead, we again do these with `if` and `else` in C++.

Also not supported is `goto case X;`. Instead, we need to create our own label and `goto` it:

```
switch (someVal)
{
    case DO_B:
        doB:
        DoB();
        break;
    case DO_A_AND_B:
        DoA();
        goto doB;
}
```

Ternary

The ternary operator in C++ is also similar to the C# version:

```
int damage = hasQuadDamage ? weapon.Damage * 4 :
weapon.Damage;
```

As in C#, this is equivalent to:

```
int damage;
if (hasQuadDamage)
    damage = weapon.Damage * 4;
else
    damage = weapon.Damage;
```

The C++ version is much looser with what we can put into the `?` and `:` parts. For example, we can throw an exception:

```
SaveHighScore() ? Unpause() : throw "Failed to save high
score";
```

In this case, the type of the expression is whatever type the non-`throw` part has: the return value of `Unpause`. We could even throw in both parts:

```
errorCode == FATAL ? throw FatalError() : throw
RecoverableError();
```

The type of the expression is `void` when we do this. Exceptions are, of course, their own category of control flow and one we'll cover more in depth later on in the book.

There are many more rules to determine the type of the ternary expression, but normally we just use the same type in both the `?` and the `:` parts like we did with the damage example. In this most typical case, the type of the ternary expression is the same as either part.

While, Do-While, Break, and Continue

`while` and `do-while` loops are essentially exactly the same as in C#:

```
while (NotAtTarget())
{
    MoveTowardTarget();
}

do
{
    MoveTowardTarget()
} while (NotAtTarget());
```

`break` and `continue` also work the same way:

```
int index = 0;
int winnerIndex = -1;
while (index < numPlayers)
{
    // Dead players can't be the winner
    // Skip the rest of the loop body by using `continue`
    if (GetPlayer(index).Health <= 0)
    {
        continue;
    }
```

```
    // Found the winner if they have at least 100 points
    // No need to keep searching, so use `break` to end
the loop
    if (GetPlayer(index).Points >= 100)
    {
        winnerIndex = index;
        break;
    }
}
if (winnerIndex < 0)
{
    DebugLog("no winner yet");
}
else
{
    DebugLog("Player", index, "won");
}
```

For

The regular three-part `for` loop is also basically the same as in C#:

```
for (int i = 0; i < numBullets; ++i)
{
    SpawnBullet();
}
```

C++ has a variant of `for` that takes the place of `foreach` in C#. It's called the "range-based for loop" and it's denoted by a colon:

```
int totalScore = 0;
for (int score : scores)
{
    totalScore += score;
}
```

It even supports an optional initialization statement like we saw with `if`:

```
int totalScore = 0;
for (int index = 0; int score : scores)
{
    DebugLog("Score at index", index, "is", score);
    totalScore += score;
```

```
    index++;
}
```

The compiler essentially converts range-based `for` loops into regular `for` loops like this:

```
int totalScore = 0;
{
    int index = 0;
    auto&& range = scores;
    auto cur = begin(range); // or range.begin()
    auto theEnd = end(range); // or range.end()
    for ( ; cur != theEnd; ++cur)
    {
        int score = *cur;
        DebugLog("Score at index", index, "is", score);
        totalScore += score;
        index++;
    }
}
```

We'll cover pointers and references soon, but for now `auto&& range = scores` is essentially making a synonym for `scores` called `range` and `*cur` is taking the value pointed at by the `cur` pointer.

There must be `begin` and `end` functions that take whatever type `scores` is, otherwise `scores` must have methods called `begin` and `end` that take no parameters. If the compiler can't find either set of `begin` and `end` functions, there will be a compiler error. Regardless of

where they are, these functions also need to return a type that can be compared for inequality (`cur != end`), pre-incremented (`++cur`), and dereferenced (`*cur`) or there will be a compiler error.

As we'll see throughout the book, there are *many* types that fit this criteria and many user-created types are designed to fit it too.

C#-Exclusive Operators

Some of C#'s control flow operators don't exist in C++ at all. First, there's no `??` or `??=` operator. The ternary operator or `if` is usually used in its place:

```
// Replacement for `??` operator
int* scores = m_Scores ? m_Scores : new Scores();

// Replacement for `??=` operator
if (!scores) scores = new Scores();
```

Second, there's no `?.` or `?[]` operator so we usually just write it out with a ternary operator for one level of indirection and `if` for more levels:

```
// Replacement for `?.` operator
int* scores = m_Scores ? m_Scores->Today : nullptr;

// Replacement for `?[]` operator
int* highScore = scores ? &scores[0] : nullptr;
```

Note that `nullptr` is equivalent to `null` in C# and is simply a null value compliant with any type of pointer but not with integers.

Return

Suitably, we end this chapter with `return`. The typical version is just like in C#:

```
int CalculateScore(int numKills, int numDeaths)
{
    return numKills*10 - numDeaths*2;
}
```

There's an alternative version where curly braces are used to more-or-less pass parameters to the constructor of the return type:

```
CircleStats GetCircleInfo(float radius)
{
    return { 2*PI*radius, PI*radius*radius };
}
```

We'll go further into constructors later in the book. For now, there's an important guarantee in the C++ language about returned objects like `CircleStats`: copy elision. This means that if the values in the curly braces are "pure," like these simple constants and primitives, then the `CircleStats` object will be initialized at the call site. This means `CircleStats` won't be allocated on the stack within `GetCircleInfo` and then copied to the call site when `GetCircleInfo` returns. This helps us avoid expensive copies when copying the return value involves copying a large amount of data such as a big array.

Conclusion

A lot of the control flow mechanisms are shared between C++ and C#. We still have `if`, `else`, `?:`, `switch`, `goto`, `while`, `do`, `for`, `foreach`/"range-based `for`", `break`, `continue`, and `return`.

C# additionally has `??`, `??=`, `?.`, and `?[]`, but C++ additionally has "init expressions" on `if`, `switch`, and range-based `for` loops, return value copy elision, and more flexibility with `?:`, `goto`, and `switch`.

These differences lead to different idioms in the way we write code in the two languages. For example, we need `begin` and `end` functions or methods in order to enable range-based `for` loops for our types in C++. If we were writing C#, we'd typically implement the `IEnumerator<T>` interface.

7. Pointers, Arrays, and Strings

Pointers

C# pointers are allowed as long as we configure the compiler to enable “unsafe” code. We then need to only use pointers within an unsafe context, such as an `unsafe` method, `unsafe` class, or `unsafe` block within a function.

C++ has no concept of “safe” or “unsafe” code. There’s no such thing as an “unsafe” context, a “safe” context, or a compiler option to enable “unsafe” code. Pointers are allowed everywhere and are commonly used in many codebases. It turns out that their syntax works very similarly to the C# pointer syntax:

```
int x = 123;

// Declare a pointer type: int* is a "pointer to an int"
// Get the address of x with &x
int* p = &x;

// Dereference the pointer to get its value
DebugLog(*p); // 123

// Dereference and assign the pointer to set its value
*p = 456;
DebugLog(x); // 456

// x->y is a convenient shorthand for (*x).y
```

```
Player* p = &localPlayer;
p->Health = 100;
```

Multiple levels of indirection are also supported by adding more `*` characters to the type:

```
int x = 123;
int* p = &x;
int** pp = &p;

DebugLog(**pp); // 123

**pp = 456;
DebugLog(x); // 456

int y = 1000;
*pp = &y;
**pp = 2000;
DebugLog(x); // 456
DebugLog(y); // 2000
```

We also have `void*`, which is a pointer to any type. A cast is required to dereference a `void*` since the compiler has no idea what type it should do the read or write on. As in C#, such a cast is not checked at runtime to ensure that the pointer really points to the type being cast to.

```
int x = 123;

// &x is an int*, but void* is compatible with all
pointer types
void* pVoid = &x;

// Cast back to int* so we can dereference
int* pInt = (int*)pVoid;
DebugLog(*pInt); // 123

// Cast to float* so we can treat the memory as though it
held another type
float* pFloat = (float*)pVoid;
*pFloat = 3.14f;
DebugLog(x); // 1078523331
```

The last line could be considered data corruption of an `int` since `3.14f` is not a valid `int`, but it's a valid way to get the bits of a `float`. This is part of the reason that these casts are unchecked.

Note that this is called "type punning" and it is technically undefined behavior, meaning the compiler might generate arbitrary machine code for this C++. At least in this simple case though, all compilers will generate the machine code that we'd expect so that we're simply treating the same memory as though it were a different type.

As in C#, pointers may be null. There are three main ways this is written in C++:

```
// nullptr is compatible with all pointer types, but not integer arithmetic
// This is generally the preferred way since C++11
int* p1 = nullptr;

// NULL is commonly defined to be zero, but works with integer arithmetic
int* p2 = NULL;

// The zero integer
int* p3 = 0;
```

Arrays

It may seem strange to see arrays lumped into the same chapter as pointers, but they're very similar in C++. Unlike in C#, arrays are not an object that's "managed" and subject to garbage collection. They are instead simply a fixed-size contiguous allocation of the same type of data:

```
// Declare an array of 3 int elements
// The elements of the array are uninitialized
int a[3];

// Initialize the first element of the array by writing
to it
a[0] = 123;

// Read the first element of the array
DebugLog(a[0]); // 123
```

When we create an array variable, it's just like we individually created its elements via variables:

```
int a0;
int a1;
int a2;
```

This means that there is no overhead for an array. It is literally just its elements. It doesn't even have an integer keeping track of its length like the `Length` field in C#. This means that the C# `stackalloc`

keyword is unnecessary as C++ arrays are already allocated on the stack when declared as local variables. Likewise, the `fixed` keyword to create a fixed-size buffer as a struct or class field is unnecessary as a C++ array's elements are already stored inside the struct or class.

There is also no bounds-checking on indexes into the array, just like indexing into a pointer in C# or C++. It's very important to be careful not to read beyond the beginning or end of the array as there's usually no way to know what data will be read or overwritten.

The lines blur even more because we can implicitly convert arrays into pointers:

```
int a[3];
a[0] = 123;

// Implicitly convert the int[3] array to an int*
// We get a pointer to the first element
int* p = a;
DebugLog(*p); // 123

// Indexing into pointers works just like in C#
DebugLog(p[0]); // 123
```

The opposite does not work though: we can't write `int b[3] = p`.

Short arrays are commonly initialized with curly braces:

```
int a[3] = { 123, 456, 789 };
DebugLog(a[0], a[1], a[2]); // 123, 456, 789
```

If we specify more elements than will fit in the array's size, we get a compiler error:

```
int a[3] = { 123, 456, 789, 1000 }; // compiler error
```

If we specify fewer elements, only the ones we specify will be initialized. Note that a trailing comma is allowed:

```
int a[3] = { 123, 456, };
DebugLog(a[0], a[1]); // 123, 456
DebugLog(a[2]); // Uninitialized. Could be anything!
```

It's common to omit the array size when using curly braces to initialize the array. This tells the compiler to count the number of elements in the curly braces and make the array that long.

```
int a[] = { 123, 456, 789 }; // The a array has 3
elements
DebugLog(a[0], a[1], a[2]); // 123, 456, 789
```

Finally, we have multi-dimensional arrays. These are arrays of arrays, both with fixed lengths. This means they are never "jagged" but always "rectangular." Just as with one-dimensional arrays, we end up with a contiguous sequence of contiguous sequences of the same type of data. There's still no overhead:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
DebugLog(a[0][0], a[0][1], a[0][2]); // 1, 2, 3
DebugLog(a[1][0], a[1][1], a[1][2]); // 4, 5, 6
```

These are implicitly converted into a pointer to the first dimension of the array:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

// Implicitly convert to a pointer to an array of 3 int
// Read the type name as "p is a pointer to an array of 3 int elements"
int (*p)[3] = a;

// Dereference that pointer to get a pointer to the first element
int* pp = *p;
for (int i = 0; i < 6; ++i)
{
    DebugLog(pp[i]); // 1, 2, 3, 4, 5, 6
}
```

Indexing into a multi-dimensional array with fewer subscripts than its dimensions just yields the remaining dimensions of the array. We can capture this in a pointer using the same implicit conversion:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int* firstRow = a[0]; // Index 1 of 2 dimensions to get the second dimension as a pointer
DebugLog(firstRow[0], firstRow[1], firstRow[2]); // 1, 2, 3
```

Pointers to Arrays and Arrays of Pointers

Sometimes we want to have a pointer to an array. This is essentially what a C# array is since we only have a reference to it, not its actual contents. Here's how we'd do that in C++:

```
int a[] = { 1, 2, 3 };

// Add a * to make this a pointer to an array instead of just an array
// This is similar to how int* is a pointer to an int
int (*p)[3] = &a;

// Dereference the pointer to get the array, which we can index into
DebugLog((*p)[0], (*p)[1], (*p)[2]); // 1, 2, 3
```

Pointers to arrays aren't supported by C# since pointers can't point to managed types like arrays.

If we want an array of pointers, just add a `*` to the type of the array element:

```
int x = 1;
int y = 2;
int z = 3;

// Add a * to int to get int*: a pointer to an int
int* a[] = { &x, &y, &z };
```

```
// Index into the array to get the pointer then
dereference it to get the int
DebugLog(*a[0], *a[1], *a[2]); // 1, 2, 3
```

Arrays of pointers are supported by C#, but the array is a managed object that we only have a reference to.

Strings

The difference with strings is similar to that of arrays. In C# we have managed `System.String` objects that are garbage-collected. In C++, we essentially have null-terminated arrays of characters:

```
// The string literal "hello" has type const char[6]
// Its contents are the characters 'h', 'e', 'l', 'l',
'o', 0
const char hello[] = "hello";

// Like any other array, it's implicitly converted a
pointer
const char* p = hello;
for (int i = 0; i < 6; ++i)
{
    DebugLog(p[i]); // h, e, l, l, o, <NUL>
}
```

We'll go into `const` more later, but for now it's just important to know that the characters of the array can't be changed. For instance, this would produce a compiler error:

```
p[0] = 'H';
```

In chapter 3, we saw that there are various kinds of character literals. The same is true for strings as each corresponds to the type of character elements in its array:

String Type	Syntax	Meaning
`char[]`	“hello”	ASCII string
`wchar_t[]`	L”hello”	“Wide character” string
`char8_t[]`	u8″hello”	UTF-8 string
`char16_t[]`	u”hello”	UTF-16 string
`char32_t[]`	U”hello”	UTF-32 string

Regardless of the character type, we can concatenate together string literals just by placing them together. No `+` operator is needed, as in C#.

```
char msg[] = "Hello, " "world!";
DebugLog(msg); // Hello, world!
```

As long as just one of the string literals has an encoding prefix, the others will get it too:

```
const char16_t msg[] = "Hello, " u"world!";
DebugLog(msg); // Hello, world!
```

Support for mixing encoding prefixes varies by compiler.

Raw strings like this are commonly used when literals suffice, such as log message text. When more advanced functionality is desired, and it very commonly is, wrapper classes such as the C++ Standard Library’s `string` or Unreal’s `FString` are used instead. We’ll go into `string` later in the book.

Pointer Arithmetic

Like in C#, arithmetic may be performed on pointers:

```
int a[3] = { 0, 0, 0 };

int* p = a; // Make p point to the first element of a
*p = 1;

p += 2; // Make p point to the third element of a
*p = 3;

--p; // Make p point to the second element of a
*p = 2;

DebugLog(a[0], a[1], a[2]); // 1, 2, 3
```

Pointers may also be compared:

```
int a[3] = { 0, 0, 0 };
int* theStart = a;
int* theEnd = theStart + 3;
while (theStart < theEnd) // Compare pointers
{
    *theStart = 1;
    theStart++;
```

```
}
DebugLog(a[0], a[1], a[2]); // 1, 1, 1
```

Recall from chapter six that this satisfies the criteria for a range-based `for` loop:

```
int a[3] = { 1, 2, 3 };
for (int val : a)
{
    DebugLog(val); // 1, 2, 3
}
```

The compiler transforms this into a normal `for` loop:

```
{
    int*&& range = a;
    int* cur = range;
    int* theEnd = range + 3;
    for ( ; cur != theEnd; ++cur)
    {
        int val = *cur;
        DebugLog(val);
    }
}
```

Note that the `begin` and `end` functions aren't required in the special case of arrays because the compiler knows the beginning and ending pointers since the size of the array is fixed at compile time.

Function Pointers

Unlike C#, in C++ we are allowed to make pointers to functions:

```
int GetHealth(Player p)
{
    return p.Health;
}

// Get a pointer to GetHealth. Syntax in three parts:
// 1) Return type: int
// 2) Pointer name: (*p)
// 3) Parameter types: (Player)
int (*p)(Player) = GetHealth;

// Calling the function pointer calls the function
int health = p(localPlayer);

DebugLog(health);
```

There are two variants of this syntax that make no difference to the functionality:

```
// Assign the address of the function instead of just its
name
int (*p)(Player) = &GetHealth;
```

```
// Dereference the function pointer before calling it
int health = (*p)(localPlayer);
```

Function pointers are commonly used like delegates in C#. They are an object that can be passed around that, when called, invokes a function. They are much more lightweight though as they are just a pointer. Delegates have much more functionality, such as the ability to add, remove, and invoke multiple functions and bind to functions of various types such as instance methods and lambdas. We'll cover how to do that in C++ later on in the book.

To make an array of function pointers, add the square brackets (`[]`) after its name like before:

```
int GetHealth(Player p)
{
    return p.Health;
}

int GetLives(Player p)
{
    return p.Lives;
}

// Array of pointers to functions that take a Player and
return an int
int (*statFunctions[])(Player) = { GetHealth, GetLives };

// Index into the array like any other array
int health = statFunctions[0](localPlayer);
```

```
DebugLog(health);
int lives = statFunctions[1](localPlayer);
DebugLog(lives);
```

Arrays of function pointers are commonly used for jump tables to replace a long chain of conditional logic with a simple index into a simple array indexed read operation.

Conclusion

C++ pointers functionality includes everything C# pointers can do and adds on the ability to create pointers to functions and pointers to any type. Arrays and strings are closely related to pointers, unlike their managed C# counterparts. Combined together, we have much enhanced functionality such as arrays of function pointers to make jump tables, a lightweight replacement for delegates, and an alternative to `stackalloc` and `fixed`-size buffers that supports any type of elements.

8. References

Pointers

As we saw last chapter, there is a lot of flexibility in pointers and their closely-associated arrays and strings. Usually, it's a lot more flexibility than we really want. In the vast majority of cases, we simply want a pointer to refer to a variable. We don't want that variable to be null, we don't intend to perform arithmetic on the pointer, and we don't want to index into it like an array. Consider a function declaration like this:

```
int GetTotalPoints(Player*);
```

This makes the reader ask themselves questions like "can the `Player` pointer be null?" The reader might also wonder "is this a single `Player` or an array of them?" and "if this is an array, how long can it be?" The answers really depend on the implementation of `GetTotalPoints`, but we don't want readers to have to guess or spend their time tracking down and reading the function definition. The function definition might not even be available, such as with a closed-source library.

Lvalue References

To address these issues, C++ introduces “references” as an alternative to pointers. A reference is like an alias to something, usually backed with a pointer in the compiled code. Here’s how one looks:

```
int x = 123;
int& r = x; // <-- reference
DebugLog(x, r); // 123, 123
```

There are a several critical aspects of this. First, the syntax for a reference is similar to a pointer except that we add a `&` instead of a `*` to the type we want to refer to: `int` in this case. We can read the resulting `int& r` as “`r` is a reference to an `int`.”

Second, we *must* initialize the reference when it’s declared. We can’t simply write `int& r;` or we’ll get a compiler error. This helps avoid undefined behavior since we can’t possibly read or write an unintialized reference.

Third, the thing we initialize the reference to must be a valid “lvalue.” This is generally thought of as “something with a name.” It includes variables and functions. It also means that a reference can never be null since everything with a name has a non-null memory address in C++.

Fourth, we don’t initialize to `&x` like we’d do with a pointer and we don’t dereference the reference with `*x`. We simply use it as an alias. Any mention of `r` is just like we mentioned `x`. References are aliases, not objects. A pointer is distinct from what it points to and can be manipulated independently, but a reference cannot. This means

there's no re-assignment of a reference because we can't actually refer to the reference that way:

```
int x = 123;
int y = 456;
int& r = x;

// This is equivalent to:
//   x = y;
// y is read and written to x
// r remains an alias of x
r = y;

DebugLog(x, r); // 456, 456
```

This is usually easier to reason about since the reference, unlike a pointer, can never change what it refers to as the program runs. We can, however, make a second reference by assigning the first reference to it:

```
int x = 123;

// Alias to x
int& r1 = x;

// This is equivalent to:
//   int& r2 = x;
// So this is also an alias to x
int& r2 = r1;
```

```
DebugLog(r1, r2); // 123, 123
x = 456;
DebugLog(r1, r2); // 456, 456
```

Because a reference isn't a distinct object, there's no such thing as a reference to a reference, pointer to a reference, or array of references:

Here are three alternate ways to initialize a reference:

```
int& r(x);
int& r = {x};
int& r{x};
```

They may also be initialized by passing them as an argument using two of the above forms:

```
void AddOne(int& val)
{
    val += 1;
}

int x = 1;

AddOne(x);
DebugLog(x); // 2
```

```
AddOne({x});
DebugLog(x); // 3
```

Likewise, returning a reference also initializes it:

```
int nextId = 0;

int& GetNextId()
{
    nextId++;
    return nextId;
}

int& id = GetNextId();
DebugLog(id); // 1
id = 0; // Reset
DebugLog(nextId); // 0
```

Now let's see a reference to a function. These look just like pointers to functions, except that there's a `&` instead of a `*`:

```
// Reference to a function that takes an int and returns
a bool
bool (&r)(int) = MyFunc;
```

We can use them like this:

```
// Function to find the index according to some matching
function
int FindIndex(int array[5], bool (&matcher)(int))
{
    for (int i = 0; i < 5; ++i)
    {
        if (matcher(array[i]))
        {
            return i;
        }
    }
    return -1;
}

bool IsEven(int val)
{
      return (val & 1) == 0;
}

// Make a reference to our matching function
bool (&isEven)(int) = IsEven;

int array[5] = { 1, 2, 3, 4, 5 };
int index = FindIndex(array, isEven); // Pass reference,
not function
DebugLog(index);
```

Because we can initialize a reference by passing an argument, there really isn't a need to explicitly make `isEven` as a local reference. Instead, we could do this:

```
// Passing the name of the function initializes the
matcher reference argument
int index = FindIndex(array, IsEven);
```

A local reference is more useful when we don't know what we want to reference at compile time and we want to use that runtime choice over and over:

```
// Decide what to alias at runtime
bool (&matcher)(int) = userWantsEvens ? IsEven : IsOdd;

// Use the result of that decision over and over
int index1 = FindIndex(array1, matcher);
int index2 = FindIndex(array2, matcher);

bool foundInBothArrays = index1 >= 0 && index2 >= 0;
```

Here's a summary of the constraints that references impose compared to pointers:

- Must be initialized when declared
- Can't be indexed into to offset a memory address
- Not subject to pointer arithmetic
- No references to references
- No pointers to references

- No arrays of references
- Can't be null
- Can't change what it aliases

That seems like a lot of lost flexibility and a lot more rules to live by, but it turns out that satisfying all of these constraints is extremely common. Aside from the last three, these are mostly the constraints that C# references impose on us and they've turned out to be quite practical. In practice, C++ references are very heavily used to succinctly convey all of these constraints to readers. Let's look once more at the function we started with, now using a reference:

```
int GetTotalPoints(Player&);
```

It's now clear that the `Player` can't be null because that's not possible with references. It's clear that that this isn't an array of `Player` objects, because that's also not possible. The `&` instead of `*` means that it's simply an alias for one non-null `Player` object.

Rvalue References

So far we've seen how references can make an alias for an "lvalue," which is something with a name. We can also make references to things without a name. These references to "rvalues" were introduced in C++11 and are used quite extensively now.

An rvalue reference has two `&` after the type it references and is initialized with something that *doesn't* have a name:

```
int&& r = 5;
```

The literal `5` doesn't have a name like a variable does. Still, we can reference it and its lifetime is extended to the lifetime of the reference so that the reference never refers to something that no longer exists. It works like this:

```
{
    // 5 is the rvalue
    // It's not just a temporary on this line
    // Its lifetime is extended to match r
    int&& r = 5;

    // 123 is the rvalue, but it's just written to x
    // 123 stops existing after the semicolon
    int x = 123;

    // Both the rvalue reference and the variable are
still readable
```

```
    DebugLog(r, x); // 5, 123

    // The temporary that r refers to is still accessible
via the alias
    r = 6;
    DebugLog(r, x); // 6, 123

    // Don't worry, we didn't overwrite the fundamental
concept of 5 :)
    DebugLog(5); // 5

// The scope that r is in ends
// r and 5 end their lifetime
// They can no longer be used
}
```

Liftime extension is much more important with structs and classes than with primitives like `int`, but the same rules apply. We'll go much more into structs and classes later in the book.

The same alternate initialization forms are allowed with rvalue references:

```
int&& r(5);
int&& r = {5};
int&& r{5};
```

We can also initialize with function arguments:

```
void PrintRange(int&& from, int&& to)
{
    for (int i = from; i <= to; ++i)
    {
        DebugLog(i);
    }
}

PrintRange(1, 3); // 1, 2, 3
```

Return values can also initialize rvalue references, but these will become “dangling” references when returning a temporary because its liftime is not extended past the end of the function call:

```
Player&& MakePlayer(int id, int health)
{
    // Create a temporary Player
    // Alias it to an rvalue reference
    // Return that alias
    return { id, health };
}

// The returned rvalue reference is "dangling"
// It refers to a temporary Player that no longer exists
// It must not be used or undefined behavior will happen
Player&& player = MakePlayer(123, 100);
```

```
// We'll get garbage when we read from it
DebugLog(player.Id, player.Health); // 17823804, 12850082
```

It's important to keep this in mind and only return rvalue references whose lifetime is already going to extend beyond the end of the function call. We'll see some techniques for doing this later on in the book.

The same constraints that apply to lvalue references apply to rvalue references:

- Must be initialized when declared
- Can't be indexed into to offset a memory address
- Not subject to pointer arithmetic
- No references to references
- No pointers to references
- No arrays of references
- Can't be null
- Can't change what it aliases

Additionally, despite the naming similarity, lvalue references are different types than rvalue references. For example, consider trying to call the above `PrintRange` function with lvalues:

```
int from = 1;
int to = 3;

// Compiler error
// Can't pass int& when int&& is required
PrintRange(from, to);
```

No other kind of initialization of an rvalue reference is possible with an lvalue, even something as simple as this:

```
int x = 123;

// Compiler error
// x is an lvalue when int&& requires an rvalue
int&& r = x;
```

We can, however, assign an rvalue reference to an lvalue reference when that rvalue reference has a name:

```
// Compiler error
// 123 is an rvalue when int& requires an lvalue
int& error = 123;

int&& rr = 123;
int& lr = rr; // rr has a name, so it's an lvalue

DebugLog(rr, lr); // 123, 123
rr = 456;
DebugLog(rr, lr); // 456, 456
```

The opposite doesn't work when the lvalue reference has a name, because that makes it not an rvalue:

```
int x = 123;
int& lr = x;
```

```
// Compiler error
// lr is an lvalue when int&& requires an rvalue
int&& rr = lr;
```

C# References

C# has several types of references. Let's compare them with C++ references.

First, there's the `ref` keyword used to pass function arguments "by reference." This is pretty close to a C++ lvalue reference as the argument must be an lvalue and acts like an alias for the variable that was passed. There are some differences though. First, C++ uses `&` instead of `ref` in the function signature and doesn't require the `ref` keyword when calling the function. Second, C# `ref` arguments can only be references to variables, not functions.

The `out` and `in` parameter modifiers are also described as enabling pass-by-reference functionality in C#. Parameters marked with `out` are also like C++ lvalue references with the additional requirement that they must be written to at least once by the function. There isn't a direct correspondence for this in C++ as the language tends to shy away from requiring at compile time that the write will be done, as is also the case with variable initialization. On the other hand, `in` parameters are essentially the same as a `const` lvalue reference in C++. We'll cover `const` more in depth later, but for now it can be thought of as like an enhanced version of `readonly` in C#.

Second, there are `ref` return values and `ref` local variables. These are also similar to C++ lvalue references since they create an alias to an lvalue. C++ uses the same `&` syntax instead of `ref` in both the function signature for `ref` returns and and variable declaration for local variables. C# also requires `ref` at the `return` statement, but C++ doesn't.

Third, there are `ref` and `readonly ref` structs in C# to force allocating them on the stack by enforcing various restrictions. This meaning of "reference" has no correlation to either lvalue or rvalue references in C++.

Fourth, and finally, there are reference types such as classes, interfaces, delegates, dynamic objects, the `object` type, and strings. All of these are “managed” types subject to garbage collection. As C++ has no “managed” types or garbage collection, there are also no reference types. Instead, references can be made to any type in C++.

The meaning of those references in C++ is different to that of C# references, though. In C#, they are somewhere in between pointers and C++ references. They’re like pointers in that they are an object, as opposed to an alias. They can be null and they and can be reassigned. They’re like references in that no pointer arithmetic is allowed and they can’t be indexed into like an array to offset a memory address.

Another major difference is that managed C# types are subject to garbage collection when there are no more references to them. This implies some behind-the-scenes tracking mechanism to know whether there are any references still available. This is very complicated, sometimes expensive, code that must be thread-safe and deal with esoteric edge cases. C++ references have no such tracking and do not imply any grand resource-management scheme. Besides lifetime extension of rvalue references, which is usually rather brief, there’s no attempt to globally manage all references for any purpose, including deallocation.

Conclusion

C++ references are similar to C++ pointers, C# pointers, and various kinds of C# references, but different in many ways from all of them. Its lvalue references are a unique way of referencing variables as well as functions. Its rvalue references are especially strange as none of these similar concepts offers anything close to the same functionality. As we go on through the book, we'll see the growing importance and common usage of both kinds of references in many other areas of the language and its Standard Library.

9. Enumerations

Unscoped Enumerations

The first kind of enumerations in C++ are called “unscoped” enumerations. This is because they don’t introduce a new scope to contain their enumerators, but instead introduce those enumerators to their surrounding scope. Consider this one:

```
enum Color
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
};

DebugLog(Red); // 0xff0000
```

This example shows several aspects of unscoped enumerations. First, defining one is very similar to in C#. We use `enum` then the enumeration’s name, and put enumerators and their values in curly braces separated by commas. Unlike in C#, we add a semicolon after the closing curly brace.

Second, we see how the `Red`, `Green`, and `Blue` enumerators are put into the surrounding scope rather than inside the `Color` enum as would have been the case in C#. This means the `DebugLog` line has `Red` in scope to read and print out.

Optionally though, we can use C++’s “scope resolution” operator to explicitly access the enumerator:

```
DebugLog(Color::Red); // 0xff0000
```

Because the name doesn't need to be used to access its enumerators, the name of the enum is itself optional:

```
enum
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
};

DebugLog(Red); // 0xff0000
```

Like in C#, the enumerators' values are optional. They even follow the same rules for default values: the first enumerator defaults to zero and subsequent enumerators default to the previous enumerator's value plus `1`:

```
enum Prime
{
    One = 1,
    Two,
    Three,
    Five = 5
};
```

```
DebugLog(One, Two, Three, Five); // 1, 2, 3, 5
```

Unlike C#, these enumerator values implicitly convert to integer types:

```
int one = One;
DebugLog(one); // 1
```

Specifically, the underlying integer type of the enum is chosen from the following list. The smallest type that can hold the largest enumerator's value is selected.

- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`

If the largest value doesn't fit in any of these types, the compiler produces an error.

For more control, the underlying integer type can be specified explicitly using the same syntax as in C#:

```
enum Color : unsigned int
{
    Red = 0xff0000,
    Green = 0x00ff00,
```

```
    Blue = 0x0000ff
};
```

Like in C#, we can cast enumerators to integers. Just note that it's undefined behavior if that integer is too small to hold the enumerator's value:

```
// OK cast to integer
int one = (int)One;
DebugLog(one); // 1

// Too big to fit in 1 byte: undefined behavior
char red = Red;
DebugLog(red); // could be anything...
```

We can also cast integers to enum variables, even if the integer value isn't one of the enumerators:

```
// OK cast to enum-typed variable
Prime prime = (Prime)3;
DebugLog(prime); // 3

// OK cast to enum-typed variable, even though not a
named enumerator
Prime prime = (Prime)4;
DebugLog(prime); // 4
```

We can also initialize them with a single integer value in curly braces as long as the integer fits in the underlying type and the underlying type has been explicitly stated:

```
Prime prime{3};
```

Note that we've used the enum name like a type here, just like we could in C#. That means we can write functions like this:

```
void OutputCharacterToLedDisplay(char ch, Color color)
{
    // ...
}
```

Passing an arbitrary integer is no longer allowed for `color`:

```
OutputCharacterToLedDisplay('J', 0xff0000); // compiler
error

OutputCharacterToLedDisplay('J', Red); // OK
```

Like with functions, enums may be declared and referenced without being defined as long as it's defined later on. When doing this, we have to specify the underlying integer type:

```
// Declare the enum
enum Color : unsigned int;
```

```
// Use the enum's name
void OutputCharacterToLedDisplay(char ch, Color color)
{
    // ...
}

// Define the enum
enum Color : unsigned int
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
};
```

Both the declaration and the definition are a type just like `int` or `float`, so they can be followed by identifiers in order to create variables:

```
// Declaration
enum Color : unsigned int red, green, blue;
red = Red;
green = Green;
blue = Blue;
DebugLog(red, green, blue); // 0xff0000, 0x00ff00,
0x0000ff

// Definition
enum Color : unsigned int
```

```
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
} red = Red, green = Green, blue = Blue;
DebugLog(red, green, blue); // 0xff0000, 0x00ff00,
0x0000ff
```

Finally, there is no special handling of bit flags like C#'s `[Flags]` attribute. Enumerators of all unscoped enumeration types may simply be used directly:

```
enum Channel
{
    RedOffset = 16,
    GreenOffset = 8,
    BlueOffset = 0
};

unsigned char GetRed(unsigned int color)
{
    return (color & Red) >> RedOffset;
}

DebugLog(GetRed(0x123456)); // 0x12
```

Scoped Enumerations

Fittingly, the other type of enumeration in C++ is called a “scoped” enumeration. As expected, this introduces a new scope which contains the enumerators. They *do not* spill out into the surrounding scope, so the “scope resolution” operator is required to access them:

```
enum class Color : unsigned int
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
};

// Compiler error: Red is not in scope
auto red = Red;

// OK, type of the red variable is Color
auto red = Color::Red;
```

There’s a lot of commonality here: `enum`, an enum name, an underlying type, enumerator names, enumerator values, the curly braces, and the semicolon at the end. The only difference is the presence of the word `class` after `enum` and before the enum’s name. This keyword tells the compiler to make a scoped enumeration instead of an unscoped one. The keyword `struct` may be used instead and has exactly the same effect as `class`.

Scoped enumerations behave mostly the same as unscoped enumerations, so we’ll just talk about the handful of differences.

First up, the name of the enum is not optional. This is because such an enum would be pretty useless since its enumerators didn't get added to the surrounding scope. Without a name to add before the scope resolution operator (`::`), there'd be no way to access them.

```
// Compiler error: no name
enum class : unsigned int
{
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff
};

// Compiler error: no way to name the enum to access its
enumerators
auto red = ???::Red;
```

Another difference is that the enumerators of a scoped enumeration don't implicitly convert to integers:

```
// Compiler error: no implicit conversion
unsigned int red = Color::Red;
```

Casting is required to convert:

```
// OK
unsigned int red = (unsigned int)Color::Red;
```

The choice of underlying type when not explicitly stated is a lot simpler, too: it's always `int`:

```
enum class Numbers
{
    // OK: 1 fits in int
    One = 1,

    // Compiler error: too big to fit in an int (assuming
int is 32-bit)
    Big = 0xffffffffffffffff
};
```

Because the underlying type is known to be `int`, the compiler can use the enumeration type without it being explicitly stated in the *declaration*:

```
// OK: underlying type not required for scoped enums
// As usual, the default underlying type is int
enum struct Prime;

// OK: definition doesn't need to specify an underlying
type either
enum struct Prime
{
    One = 1,
    Two,
    Three,
```

```
    Five = 5
};

// OK: definition is allowed to specify an underlying
type as long as it's int
enum struct Prime : int
{
    One = 1,
    Two,
    Three,
    Five = 5
};
```

Those are actually all the differences between the two kinds of enumerations. The following table compares and contrasts them with each other and C# enumerations:

Aspect	**Example**	**Unscoped**	**Scoped**	**C#**
Initialization of enumerators	`One = 1`	Optional	Optional	Optional
Casting enumerators to integers	`int one = (int)One;`	Yes	Yes	Yes
Casting integers to enumerators	`Prime p = (Prime)4;`	Yes	Yes	Yes
Name	`enum Prime {};`	Optional	Required	Required

Aspect	Example	Unscoped	Scoped	C#
Implicit enumerators-to-integer conversion	C++: `int one = Prime::One` C#: `int one = Prime::One`	Yes	No	No
Scope resolution operator	C++: `Prime::One` C#: `Prime.One`	Optional	Required	Required
Implicit underlying type	`enum E {};`	`int` or larger	`int`	`int`
Underlying type required for declaration	`enum E;`	Yes	No	N/A (no declarations)
Initialization from integer	C++: `Prime p{4}` C#: `Prime p = 4`	Yes	Yes	No
Immediate variables	`enum Prime {} p;`	Yes	Yes	No
Requirement to use bitwise operators		None	Casting	None (`[Flags]` optional)

Conclusion

Of the two kinds of enumerations in C++, scoped enumerations are definitely closest to C# enumerations. Still, C++ has unscoped enumerations and they are commonly used. It's important to know the differences between them, scoped enumerations, and C# enumerations as they have a number of subtle differences to keep in mind.

10. Struct Basics

Declaration and Definition

Just like with functions and enumerations, structs may be declared and defined separately:

```
// Declaration
struct Vec3;

// Definition
struct Vec3
{
    float x;
    float y;
    float z;
};
```

Notice how struct declarations and definitions looks pretty similar to enumeration declarations and definitions. We use the `struct` keyword, give it a name, add curly braces to hold its contents, then finish up with a semicolon.

We can create struct variables just like we create primitive and enumeration variables:

```
Vec3 vec;
```

As with primitives and enumerations, this variable is uninitialized. Initialization of structs is a surprisingly complex topic compared to C# and we'll cover it in depth later on in the book. For now, let's just initialize by individually setting each of the struct's data members. That's the C++ term for the equivalent of fields in C#. They're also commonly called "member variables." To do this, we use the `.` operator just like in C#:

```
Vec3 vec;
vec.x = 1;
vec.y = 2;
vec.z = 3;

DebugLog(vec.x, vec.y, vec.z); // 1, 2, 3
```

We can also initialize the data members in the struct definition with either `=x` or `{x}`:

```
struct Vec3
{
    float x = 1;
    float y{2};
    float z = 3;
};

Vec3 vec;
DebugLog(vec.x, vec.y, vec.z); // 1, 2, 3
```

As with enumerations, we can also declare variables between the closing curly brace and the semicolon of a definition:

```
struct Vec3
{
    float x;
    float y;
    float z;
} v1, v2, v3;
```

This is sometimes used when omitting the name of the struct. This anonymous struct has no name we can type out, but it can be used all the same in a similar way to C# tuples (`(string Name, int Year) t = ("Apollo 11", 1969);`):

```
// Anonymous struct with immediate variable
struct
{
    const char16_t* Name;
    int32_t Year;
} moonMission;

// Variables of this type can be used just like named
struct types
moonMission.Name = u"Apollo 11";
moonMission.Year = 1969;
DebugLog(moonMission.Name, moonMission.Year);
```

Because an anonymous struct can only be used via immediate variables, declaring one without any immediate variables isn't allowed:

```
// Compiler error: anonymous struct requires at least one immediate variable
struct { float x; };
```

Like with enumerations whose underlying type isn't in the declaration, the compiler doesn't know the size of a struct after it's declared. The definition is required to know its size, so a declared struct can't be used to create a variable or define a function using the struct type as an parameter or return value:

```
// Declaration
struct Vec3;

// Compiler error: can't create a variable before definition
Vec3 v;

// Compiler error: can't take a function parameter before definition
float GetMagnitudeSquared(Vec3 vec)
{
    return 0;
}

// Compiler error: can't return a function return value
```

```
before definition
Vec3 MakeVec(float x, float y, float z)
{
    // Compiler error: can't create a variable before
definition
    Vec3 v;

    // Compiler error: can't return a struct before
definition
    return v;
}
```

This also means that we can't declare immediate variables after a struct declaration:

```
// Compiler error: can't create a variable before
definition
struct Vec3 v1, v2, v3;
```

We can, however, use pointers and references to the struct since they don't depend on its size:

```
// Declaration
struct Vec3;

// Pointer
Vec3* p = nullptr;
```

```
// lvalue reference
float GetMagnitudeSquared(Vec3& vec)
{
    return 0;
}

// rvalue reference
float GetMagnitudeSquared(Vec3&& vec)
{
    return 0;
}
```

To access the fields of a pointer, we can either dereference with `*p` and then use `.x` or use the shorthand `p->x`. Both are exactly equivalent to struct pointers in C#. With lvalue or rvalue references, we just use `.` because they are essentially just aliases to a variable, not a pointer.

```
// Variable
Vec3 vec;
vec.x = 1;
vec.y = 2;
vec.z = 3;

// Pointer
Vec3* p = &vec;
p->x = 10;
p->y = 20;
(*p).z = 30; // Alternate version of p->z
```

```
// lvalue reference
float GetMagnitudeSquared(Vec3& vec)
{
    return vec.x*vec.x + vec.y*vec.y + vec.z*vec.z;
}

// rvalue reference
float GetMagnitudeSquared(Vec3&& vec)
{
    return vec.x*vec.x + vec.y*vec.y + vec.z*vec.z;
}
```

Layout

Like in C#, the data members of a struct are grouped together in memory. Exactly how they're laid out in memory isn't defined by the C++ Standard though. Each compiler will lay out the data members as appropriate for factors such as the CPU architecture being compiled for.

This is similar to the default struct layout in C#, which behaves as though `[StructLayout(LayoutKind.Auto)]` were explicitly added. There is no `[StructLayout]` attribute in C++, but compiler-specific preprocessor directives are available to gain similar levels of control.

That said, compilers virtually always lay out the data members in a predictible pattern. Each is placed sequentially in the same order as written in the source code. Padding is placed between the data members according to the alignment requirements of the data types, which varies by CPU architecture. For example:

```
struct Padded  // Takes up 8 bytes
{
    int8_t a;  // Takes up 1 byte
               // Padding of 3 bytes
    int32_t b; // Takes up 4 bytes
};
```

The C++ Standard does make one guarantee though: a "standard layout." This says that if two structs start with the same sequence of data types then those data members will be laid out the same. There are complex exceptions to this, but it'll hold for most normal use cases like these. This means we can safely reinterpret some common struct types:

```
struct Vec3
{
    float x;
    float y;
    float z;
};

struct Quat
{
    // Starts with the same three floats as Vec3
    float x;
    float y;
    float z;

    // Not in common. May be placed anywhere later in
memory.
    float w;
};

// Reinterpret Vec3 as Quat
Vec3 vec;
Vec3* pVec = &vec;
Quat* pQuat = (Quat*)pVec;

// Safe to use the three starting data members because
types match
pQuat->x = 1;
pQuat->y = 2;
```

```
pQuat->z = 3;

// Definitely not safe to use the last data member
// Vec3 doesn't have a fourth float
// This is undefined behavior and probably corrupts
memory
pQuat->w = 4;

DebugLog(pQuat->x, pQuat->y, pQuat->z); // 1, 2, 3
DebugLog(pVec->x, pVec->y, pVec->z); // 1, 2, 3
DebugLog(vec.x, vec.y, vec.z); // 1, 2, 3
```

Bit Fields

In C#, we can manually create bit fields but C++ supports them natively for all integer data members including `bool`. This allows us to specify how many bits of memory a data member occupies:

```
struct Player
{
    bool IsAlive : 1;
    uint8_t Lives : 3;
    uint8_t Team : 2;
    uint8_t WeaponID : 2;
};
```

This struct takes up just one byte of memory because the sum of its bit fields' sizes is `8`. Normally it would have taken up `4` bytes since each data member would take up a whole byte of its own.

We can access these data members just like normal:

```
Player p;
p.IsAlive = true;
p.Lives = 5;
p.Team = 2;
p.WeaponID = 1;

DebugLog(p.IsAlive, p.Lives, p.Team, p.WeaponID); //
true, 5, 2, 1
```

The compiler will, as always, generate CPU instructions specific to the arhitecture being compiled for and depending on settings such as optimization level. Generally though, the instructions will read one or more bytes containing the desired bits, use a bit mask to remove the other bits that were read, and shift the desired bits to the least-significant part of the data member's type. Writing to a bit field is a similar process.

As of C++20, bit fields may be initialized in the struct definition just like other data members:

```
struct Player
{
    bool IsAlive : 1 = true;
    uint8_t Lives : 3 {5};
    uint8_t Team : 2 {2};
    uint8_t WeaponID : 2 = 1;
};

DebugLog(p.IsAlive, p.Lives, p.Team, p.WeaponID); //
true, 5, 2, 1
```

Note that the size of a bit field may be larger than the stated type:

```
struct SixtyFourKilobits
{
    uint8_t Val : 64*1024;
};
```

The size of `Val` and the struct itself is 64 kilobits, but `Val` is still used just like an 8-bit integer.

Bit fields may also be unnamed:

```
struct FirstLast
{
    uint8_t First : 1; // First bit of the byte
    uint8_t : 6;       // Skip six bits
    uint8_t Last : 1;  // Last bit of the byte
};
```

Unnamed bit fields can also have zero size, which tells the compiler to put the next data member on the next byte it aligns to:

```
struct FirstBitOfTwoBytes
{
    uint8_t Byte1 : 1;  // First bit of the first byte
    uint8_t : 0;        // Skip to the next byte
    uint8_t Byte2 : 1;  // First bit of the second byte
};
```

Finally, since bit fields don't necessarily start at the beginning of a byte we can't take their memory address:

```
FirstBitOfTwoBytes x;
```

```
// Compiler error: can't take the address of a bit field
uint8_t* p = &x.Byte1;
```

Static Data Members

Like static fields in C#, data members may be static in C++:

```
struct Player
{
    int32_t Score;
    static int32_t HighScore;
};
```

The meaning is the same as in C#. Each `Player` object doesn't have a `HighScore` but rather there is one `HighScore` for all `Player` objects. Because it's bound to the struct type, not an instance of the struct, we use the scope resolution operator (`::`) as we did with scoped enumerations to access the data member:

```
Player::HighScore = 0;
```

What we put inside the struct definition is actually just a declaration of a variable, so we still need to define it outside the struct:

```
struct Player
{
    int32_t Score;
    static int32_t HighScore; // Declaration
};

// Definition
```

```
int32_t Player::HighScore;

// Incorrect definition
// This just creates a new HighScore variable
// We need the "Player::" part to refer to the
declaration
int32_t HighScore;
```

This also gives us an opportunity to initialize the variable:

```
int32_t Player::HighScore = 0;
```

Because the static data member inside the struct definition is just a declaration, it can use other types that haven't yet been defined as long as they're defined by the time we define the static data member:

```
// Declaration
struct Vec3;

struct Player
{
    int32_t Health;

    // Declaration
    static Vec3 Fastest;
};

// Definition
```

```
struct Vec3
{
    float x;
    float y;
    float z;
};

// Definition
Vec3 Player::Fastest;
```

If the static data member is `const`, we can initialize it inline. We'll go over `const` later in the book, but for now it's similar to `readonly` in C#.

```
struct Player
{
    int32_t Health;
    const static int32_t MaxHealth = 100;
};
```

We're still allowed to put the definition outside the struct, but it's optional to do so. If we do, we can only put the initialization in one of the two places:

```
// Option 1: initialize in the struct definition
struct Player
{
    int32_t Health;
```

```
    const static int32_t MaxHealth = 100;
};
const int32_t Player::MaxHealth;

// Option 2: initialize outside the struct definition
struct Player
{
    int32_t Health;
    const static int32_t MaxHealth;
};
const int32_t Player::MaxHealth = 100;

// Compiler error if initializing in both places
struct Player
{
    int32_t Health;
    const static int32_t MaxHealth = 100;
};
const int32_t Player::MaxHealth = 100;
```

Static data members may also be `inline`, much like with global variables:

```
struct Player
{
    int32_t Health;
    inline static int32_t MaxHealth = 100;
};
```

In this case, we *can't* put a definition outside of the struct:

```
struct Player
{
    int32_t Health;
    inline static int32_t MaxHealth = 100;
};

// Compiler error: can't define outside the struct
int32_t Player::MaxHealth;
```

Lastly, static data members can't be bit fields. This would make no sense since they're not part of instances of the struct and aren't even necessarily located together in memory with other static data members of the struct:

```
struct Flags
{
    // All of these are compiler errors
    // Static data members can't be bit fields
    static bool IsStarted : 1;
    static bool WonGame : 1;
    static bool GotHighScore : 1;
    static bool FoundSecret : 1;
    static bool PlayedMultiplayer : 1;
    static bool IsLoggedIn : 1;
    static bool RatedGame : 1;
```

```
    static bool RanBenchmark : 1;
};
```

To work around this, make a struct with non-static bit fields and another struct with a static instance of the first struct:

```
struct FlagBits
{
    bool IsStarted : 1;
    bool WonGame : 1;
    bool GotHighScore : 1;
    bool FoundSecret : 1;
    bool PlayedMultiplayer : 1;
    bool IsLoggedIn : 1;
    bool RatedGame : 1;
    bool RanBenchmark : 1;
};

struct Flags
{
    static FlagBits Bits;
};

FlagBits Flags::Bits;

Flags::Bits.WonGame = true;
```

Disallowed Data Members

C++ forbids using some kinds of data members in structs. First, `auto` is not allowed for the data type:

```
struct Bad
{
    // Compiler error: auto isn't allowed even if we
initialize it inline
    auto Val = 123;
};
```

An exception to this rule is when the data member is both `static` and `const`:

```
struct Good
{
    // OK since data member is static and const
    static const auto Val = 123;
};
```

Next, while `register` is only deprecated for other kinds of variables, it's illegal for data members:

```
struct Bad
{
    // Compiler error: data members can't be register
```

```
variables
    register int Val = 123;
};
```

This is also true for other storage class specifiers like `extern`:

```
struct Bad
{
    // Compiler error: data members can't be extern
variables
    extern int Val = 123;
};
```

The entire struct can be declared with either storage class specifier instead:

```
struct Good
{
    uint8_t Val;
};

register Good r;
extern Good e;
```

While we saw above that declared types that aren't yet defined can be used for static data members, this is not the case for non-static data members:

```
struct Vec3;

struct Bad
{
    // Compiler error: Vec3 isn't defined yet
    Vec3 Pos;
};
```

As with other variables of types that are declared but not yet defined, we *are* allowed to have pointers and references:

```
struct Vec3;

struct Good
{
    // OK to have a pointer to a type that's declared but
not yet defined
    Vec3* PosPointer;

    // OK to have an lvalue to a type that's declared but
not yet defined
    Vec3& PosLvalueReference;

    // OK to have an rvalue to a type that's declared but
not yet defined
    Vec3&& PosRvalueReference;
};
```

Nested Types

C++ allows us to nest types within structs just like we can in C#. Let's start with a scoped enumeration:

```
struct Character
{
    enum struct Type
    {
        Player,
        NonPlayer
    };

    Type Type;
};

Character c;
c.Type = Character::Type::Player;
```

Note how we use `Character::Type` to refer to `Type` within `Character` and then `::Player` to refer to an enumerator within `Type`.

Also note how we can have both a `Type` enumeration and a `Type` data member. The two are disambiguated by the operator used to access the content of the struct:

```
Character c;
Character* p = &c;
Character& r = c;
```

```
// . operator means "access data member"
auto t = c.Type;
t = r.Type;

// -> operator means "dereference pointer then access
data member"
t = p->Type;

// :: operator means "get something scoped to the type"
Character::Type t2;
```

Ambiguity arises if the data member is static and has the same name as a nested type:

```
struct Character
{
    enum struct Type
    {
        Player,
        NonPlayer
    };

    static Type Type;
};

// Compiler error: Character::Type is ambiguous
// It could be either the scoped enumeration or the
```

```
static data member
Character::Type Character::Type =
Character::Type::Player;
```

We can also nest unscoped enumerations:

```
struct Character
{
    enum Type
    {
        Player,
        NonPlayer
    };

    Type Type;
} c;

// Optionally specify the unscoped enumeration type name
c.Type = Character::Type::Player;

// Or don't specify it
// Enumerators are added to the surrounding scope: the
struct
c.Type = Character::Player;
```

Finally, we can nest structs within structs. As with enumerations, this can be used to contextualize them such as to clean up our `Flags` example above:

```
struct Flags
{
    struct FlagBits
    {
        bool IsStarted : 1;
        bool WonGame : 1;
        bool GotHighScore : 1;
        bool FoundSecret : 1;
        bool PlayedMultiplayer : 1;
        bool IsLoggedIn : 1;
        bool RatedGame : 1;
        bool RanBenchmark : 1;
    };

    static FlagBits Bits;
};

Flags::FlagBits Flags::Bits;
```

We can combine this with anonymous structs to eliminate some of the verbosity. If we do, we'll need to use decltype in order to state the type of the static variable when we define it outside the struct since we didn't give it an explicit name:

```
struct Flags
{
    // Unnamed struct with bit fields
    // The data member Bits is static
```

```
    static struct
    {
        bool IsStarted : 1;
        bool WonGame : 1;
        bool GotHighScore : 1;
        bool FoundSecret : 1;
        bool PlayedMultiplayer : 1;
        bool IsLoggedIn : 1;
        bool RatedGame : 1;
        bool RanBenchmark : 1;
    } Bits;
};

// The unnamed struct has no name we can just type
// Use decltype to refer to its type
decltype(Flags::Bits) Flags::Bits;

Flags::Bits.WonGame = true;
```

Of course we can continue to nest structs infinitly within other structs, but it's generally a good idea to keep it to two or three levels and avoid resorting to anything like this:

```
struct S1
{
    struct S2
    {
        struct S3
```

```
        {
            struct S4
            {
                struct S5
                {
                    uint8_t Val;
                };
            };
        };
    };
};

S1::S2::S3::S4::S5 s;
s.Val = 123;
```

Conclusion

We're only just scratching the surface of C++ structs and already they have quite a few more advanced features than their C# counterparts:

Feature	**Example**
Split declaration and definition	`struct S; struct S{};`
Inline data member initializers	`struct S {int X=1, int Y=2};`
Bit fields	`struct S {bool a:1; bool :6; bool b:1;};`
Immediate variables	`struct S {} s;`
Anonymous structs	`struct {float X; float Y;} pos2;`
References to structs	`struct S {} s; S& lr = s; S&& rr = S();`
Automatic data member typing	`struct S {static const auto X=1;};`
Shared nested type and data member name	`struct S {enum E{}; E E;};`

11. Struct Functions

Member Functions

As in C#, structs in C++ may contain functions. These are called “methods” in C# and “member functions” in C++. They look and work essentially the same as in C#:

```
struct Vector2
{
    float X;
    float Y;

    float SqrMagnitude()
    {
        return this->X*this->X + this->Y*this->Y;
    }
};
```

Member functions are implicitly passed a pointer to the instance of the struct they’re contained in. In this case, it’s type is `Vector2*`. Other than using `this->X` or `(*this).X` instead of just `this.X`, its usage is the same as in C#. It is also optional, again like C#:

```
float SqrMagnitude()
{
    return X*X + Y*Y;
}
```

Unlike C#, but in keeping with other C++ functions and with data member initialization, we can split the function's declaration and definition. If we do so, we need to place the definition outside the class:

```
struct Vector2
{
    float X;
    float Y;

    // Declaration
    float SqrMagnitude();
};

// Definition
float Vector2::SqrMagnitude()
{
    return X*X + Y*Y;
}
```

Notice that when we do this we need to specify where the function we're defining is declared. We do this by prefixing `Vector2::` to the beginning of the function's name.

It's very common to only declare member function declarations in a struct definition. That struct definition is typically put in a header file (e.g. `Vector2.h`) and the member function definitions are put into a translation unit (e.g. `Vector2.cpp`). This cuts down compile times by only compiling the member function definitions once while allowing the member functions to be called by any file that `#include`s the header file with the member function declaration.

Now that we have a member function, let's call it!

```
Vector2 v;
v.X = 2;
v.Y = 3;
float sqrMag = v.SqrMagnitude();
DebugLog(sqrMag); // 13
```

Calling the member function works just like in C# and lines up with how we access data members. If we had a pointer, we'd use `->` instead of `.`:

```
Vector2* p = &v;
float sqrMag = p->SqrMagnitude();
```

All the rules that apply to the global functions we've seen before apply to member functions. This includes support for overloading:

```
struct Weapon
{
    int32_t Damage;
};

struct Potion
{
    int32_t HealAmount;
};
```

```
struct Player
{
    int32_t Health;

    void Use(Weapon weapon, Player& target)
    {
        target.Health -= weapon.Damage;
    }

    void Use(Potion potion)
    {
        Health += potion.HealAmount;
    }
};

Player player;
player.Health = 50;

Player target;
target.Health = 50;

Weapon weapon;
weapon.Damage = 10;

player.Use(weapon, target);
DebugLog(target.Health); // 40

Potion potion;
```

```
potion.HealAmount = 20;

player.Use(potion);
DebugLog(player.Health); // 70
```

Remember that member functions take an implicit `this` argument. We need to be able to overload based on that argument in addition to all the explicit arguments. It doesn't make sense to overload on the type of `this`, but C++ does provide us a way to overload based on whether the member function is being called on an lvalue reference or an rvalue reference:

```
struct Test
{
    // Only allow calling this on lvalue objects
    void Log() &
    {
        DebugLog("lvalue-only");
    }

    // Only allow calling this on rvalue objects
    void Log() &&
    {
        DebugLog("rvalue-only");
    }

    // Allow calling this on lvalue or rvalue objects
    // Note: not allowed if either of the above exist
    void Log()
```

```
    {
        DebugLog("lvalue or rvalue");
    }
};

// Pretend the "lvalue or rvalue" version isn't
defined...

// 'test' has a name, so it's an lvalue
Test test;
test.Log(); // lvalue-only

// 'Test()' doesn't have a name, so it's an rvalue
Test().Log(); // rvalue-only
```

We'll go more into initialization of structs soon, but for now `Test()` is a way to create an instance of a `Test` struct.

Finally, member functions may be static with similar syntax and meaning to C#:

```
struct Player
{
    int32_t Health;

    static int32_t ComputeNewHealth(int32_t oldHealth,
int32_t damage)
    {
        return damage >= oldHealth ? 0 : oldHealth -
```

```
damage;
    }
};
```

To call this, we refer to the member function using the struct type rather than an instance of the type. This is just like in C#, except that we use `::` instead of `.` as is normal for referring to the contents of a type in C++:

```
DebugLog(Player::ComputeNewHealth(100, 15)); // 85
DebugLog(Player::ComputeNewHealth(10, 15)); // 0
```

Since static member functions don't operate on a particular struct object, they have no implicit `this` argument. This makes them compatible with regular function pointers:

```
// Get a function pointer to the static member function
int32_t (*cnh)(int32_t, int32_t) =
Player::ComputeNewHealth;

// Call it
DebugLog(cnh(100, 15)); // 85
DebugLog(cnh(10, 15)); // 0
```

Overloaded Operators

Both C# and C++ allow a lot of operator overloading, but there are also quite a few differences. Let's start with something basic:

```
struct Vector2
{
    float X;
    float Y;

    Vector2 operator+(Vector2 other)
    {
        Vector2 result;
        result.X = X + other.X;
        result.Y = Y + other.Y;
        return result;
    }
};

Vector2 a;
a.X = 2;
a.Y = 3;

Vector2 b;
b.X = 10;
b.Y = 20;

Vector2 c = a + b;
```

```
DebugLog(a.X, a.Y); // 2, 3
DebugLog(b.X, b.Y); // 10, 20
DebugLog(c.X, c.Y); // 12, 23
```

Here we see an overloaded binary `+` operator. It looks just like a member function except it's name is `operator+` instead of an identifier like `Use`. This is different from C# where the overloaded operator would be `static` and therefore need to take two parameters. If a C#-style static approach is desired, the overloaded operator may be declared outside the struct instead:

```
struct Vector2
{
    float X;
    float Y;
};

Vector2 operator+(Vector2 a, Vector2 b)
{
    Vector2 result;
    result.X = a.X + b.X;
    result.Y = a.Y + b.Y;
    return result;
}

// (usage is identical)
```

Another difference is that the overloaded operator may be called directly by using `operator+` in place of a member function name

when its defined inside the struct:

```
Vector2 d = a.operator+(b);
DebugLog(d.X, d.Y);
```

The following table compares which operators may be overloaded in the two languages:

Operator	**C++**	**C#**
`+x`	Yes	Yes
`-x`	Yes	Yes
`!x`	Yes	Yes
`~x`	Yes	Yes
`x++`	Yes	Yes, but same for `x++` and `++x`
`x--`	Yes	Yes, but same for `x--` and `--x`
`++x`	Yes	Yes, but same for `x++` and `++x`
`--x`	Yes	Yes, but same for `x--` and `--x`
`true`	N/A	Yes
`false`	N/A	Yes
`x + y`	Yes	Yes
`x - y`	Yes	Yes

Operator	C++	C#
`x * y`	Yes	Yes
`x / y`	Yes	Yes
`x % y`	Yes	Yes
`x ^ y`	Yes	Yes
`x && y`	Yes	Yes
`x | y`	Yes	Yes
`x = y`	Yes	No
`x < y`	Yes	Yes, requires `>` too
`x > y`	Yes	Yes, requires `<` too
`x += y`	Yes	No, implicitly uses `+`
`x -= y`	Yes	No, implicitly uses `-`
`x *= y`	Yes	No, implicitly uses `*`
`x /= y`	Yes	No, implicitly uses `/`
`x %= y`	Yes	No, implicitly uses `%`
`x ^= y`	Yes	No, implicitly uses `^`
`x &= y`	Yes	No, implicitly uses `&`
`x |= y`	Yes	No, implicitly uses `|`
`x << y`	Yes	Yes
`x >> y`	Yes	Yes
`x >>= y`	Yes	No, implicitly uses `>>`

Operator	C++	C#
`x <<= y`	Yes	No, implicitly uses `<<`
`x == y`	Yes	Yes, requires `!=` too
`x != y`	Yes	Yes, requires `==` too
`x <= y`	Yes	Yes, requires `>=` too
`x >= y`	Yes	Yes, requires `<=` too
`x <=> y`	Yes	N/A
`x && y`	Yes, without short-circuiting	No, implicitly uses `true` and `false`
`x || y`	Yes, without short-circuiting	No, implicitly uses `true` and `false`
`x, y`	Yes, without left-to-right sequencing	No
`x->y`	Yes	No
`x(x)`	Yes	No
`x[i]`	Yes	No, indexers instead
`x?.[i]`	N/A	No, indexers instead
`x.y`	No	No
`x?.y`	N/A	No
`x::y`	No	No
`x ? y : z`	No	No
`x ?? y`	N/A	No

Operator	C++	C#
`x ??= y`	N/A	No
`x..y`	N/A	No
`=>`	N/A	No
`as`	N/A	No
`await`	N/A	No
`checked`	N/A	No
`unchecked`	N/A	No
`default`	N/A	No
`delegate`	N/A	No
`is`	N/A	No
`nameof`	N/A	No
`new`	Yes	No
`sizeof`	No	No
`stackalloc`	N/A	No
`typeof`	N/A	No

As in C#, the C++ language puts little restriction on the parameters, return values, and functionality of overloaded operators. Instead, both languages rely on conventions. As such, it'd be legal but very strange to implement an overloaded operator like this:

```
struct Vector2
{
    float X;
    float Y;

    int32_t operator++()
    {
        return 123;
    }
};

Vector2 a;
a.X = 2;
a.Y = 3;
int32_t res = ++a;
DebugLog(res); // 123
```

One particularly interesting operator in the above table is `x <=> y`, introduced in C++20. This is called the "three-way comparison" or "spaceship" operator. This can be used in general, without operator overloading, like so:

```
auto res = 1 <=> 2;

if (res < 0)
{
    DebugLog("1 < 2"); // This gets called
}
```

```
else if (res == 0)
{
    DebugLog("1 == 2");
}
else if (res > 0)
{
    DebugLog("1 > 2");
}
```

This is like most sort comparators where a negative value is returned to indicate that the first argument is less than the second, a positive value to indicate greater, and zero to indicate equality. The exact type returned isn't specified other than that it needs to support these three comparisons.

While it can be used directly like this, it's especially valuable for operator overloading as it implies a canonical implementation of all the other comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. That allows us to write code that either uses the three-way comparison operator directly or indirectly:

```
struct Vector2
{
    float X;
    float Y;

    float SqrMagnitude()
    {
        return this->X*this->X + this->Y*this->Y;
    }
```

```
    float operator<=>(Vector2 other)
    {
        return SqrMagnitude() - other.SqrMagnitude();
    }
};

int main()
{
    Vector2 a;
    a.X = 2;
    a.Y = 3;

    Vector2 b;
    b.X = 10;
    b.Y = 20;

    // Directly use <=>
    float res = a <=> b;
    if (res < 0)
    {
        DebugLog("a < b");
    }

    // Indirectly use <=>
    if (a < b)
    {
        DebugLog("a < b");
```

```
        }
    }
```

Conclusion

This chapter we've seen C++'s version of methods, called member functions, and overloaded operators. Member functions are quite similar to their C# counterparts, but do have differences such as an optional declaration-definition split, overloading based on lvalue and rvalue objects, and conversion to function pointers.

Overloaded operators also have their similarities and differences to C#. In C++, they may be placed inside the struct and used like a non-static member function or outside the struct and used like a static one. When inside the struct, they can be called explicitly like with `x.operator+(10)`. Quite a few more operators may be overloaded, and often with finer-grain control. Lastly, the three-way comparison ("spaceship") operator allows for removing a lot of boilerplate when overloading comparisons.

12. Constructors and Destructors

General Constructors

First things first, we're not going to deeply discuss actually calling any of these constructors in this chapter. Initialization is a complex topic that requires a full chapter of its own. So we'll write the constructors in this chapter's chapter and call them in next chapter's chapter.

Basic C++ constructors are so similar to constructors in C# that the syntax is identical and has the same meaning!

```
struct Vector2
{
    float X;
    float Y;

    Vector2(float x, float y)
    {
        X = x;
        Y = y;
    }
};
```

Anything more advanced than this simple example is going to diverge a *lot* between the languages. First, as with member functions, we can split the constructor *declaration* from the *definition* by placing the definition outside the struct. This is commonly used to

put the declaration in a header file (`.h`) and the definition in a translation unit (`.cpp`) to reduce compile times.

```
struct Vector2
{
    float X;
    float Y;

    Vector2(float x, float y);
};

Vector2::Vector2(float x, float y)
{
    X = x;
    Y = y;
}
```

C++ provides a way to initialize data members before the function body runs. These are called “initializer lists.” They are placed between the constructor’s signature and its body.

```
struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;

    Ray2(float originX, float originY, float directionX,
float directionY)
```

```
        : Origin(originX, originY), Direction{directionX,
directionY}
    {
    }
};
```

The initializer list starts with a `:` and then lists a comma-delimited list of data members. Each has its initialization arguments in either parentheses (`Origin(originX, originY)`) or curly braces (`Direction{directionX, directionY}`). The order doesn't matter since the order the data members are declared in the struct is always used.

We can also use an initializer list to initialize primitive types. Here's an alternate version of `Vector2` that does that:

```
struct Vector2
{
    float X;
    float Y;

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};
```

The initializer list overrides data members' default initializers. This means the following version of `Vector2` has the `x` and `y` arguments initialized to the constructor arguments, not `0`:

```
struct Vector2
{
    float X = 0;
    float Y = 0;

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};
```

An initializer list can also be used to call another constructor, which helps reduce code duplication and “helper” functions (typically named `Init` or `Setup`). Here’s one in `Ray2` that defaults the origin to `(0, 0)`:

```
struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;

    Ray2(float originX, float originY, float directionX,
float directionY)
        : Origin(originX, originY), Direction{directionX,
directionY}
    {
    }
```

```
    Ray2(float directionX, float directionY)
        : Ray2(0, 0, directionX, directionY)
    {
    }
};
```

If an initializer list calls another constructor, it can *only* call that constructor. It can't also initialize data members:

```
Ray2(float directionX, float directionY)
    // Compiler error: constructor call must stand alone
    : Origin(0, 0), Ray2(0, 0, directionX, directionY)
{
}
```

Default Constructors

A “default constructor” has no parameters. In C#, the default constructor for a struct is always available and can’t even be defined by our code. C++ does allow us to write default constructors for our structs:

```
struct Vector2
{
    float X;
    float Y;

    Vector2()
    {
        X = 0;
        Y = 0;
    }
};
```

In C#, this constructor is always generated for all structs by the compiler. It simply initializes all fields to their default values, `0` in this case.

```
// C#
Vector2 vecA = new Vector2();     // 0, 0
Vector2 vecB = default(Vector2); // 0, 0
Vector2 vecC = default;           // 0, 0
```

C++ compilers also generate a default constructor for us. Like C#, it also initializes all fields to their default values.

C++ structs also behave in the same way that C# classes behave: if a struct defines a constructor then the compiler won't generate a default constructor. That means that this version of `Vector2` doesn't get a compiler-generated default constructor:

```
struct Vector2
{
    float X;
    float Y;

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};
```

If we try to create an instance of this `Vector2` without providing the two `float` arguments, we'll get a compiler error:

```
// Compiler error: no default constructor so we need to
provide two floats
Vector2 vec;
```

If we want to get the default constructor back, we have two options. First, we can define it ourselves:

```
struct Vector2
{
    float X;
    float Y;

    Vector2()
    {
    }

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};
```

Second, we can use `= default` to tell the compiler to generate it for us:

```
struct Vector2
{
    float X;
    float Y;

    Vector2() = default;

    Vector2(float x, float y)
        : X(x), Y(y)
```

```
    {
    }
};
```

We can also put `= default` outside the struct, usually in a translation unit (`.cpp` file):

```
struct Vector2
{
    float X;
    float Y;

    Vector2();

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};

Vector2::Vector2() = default;
```

Sometimes we want to do the reverse and stop the compiler from generating a default constructor. Normally we do this by writing a constructor of our own, but if we don't want to do that then we can use `= delete`:

```
struct Vector2
{
    float X;
    float Y;

    Vector2() = delete;
};
```

This *can't* be put outside the struct:

```
struct Vector2
{
    float X;
    float Y;

    Vector2();
};

// Compiler error
// Must be inside the struct
Vector2::Vector2() = delete;
```

If there's no default constructor, either generated by the compiler or written by hand, then the compiler also won't generate a default constructor for structs that have that kind of data member:

```
// Compiler doesn't generate a default constructor
// because Vector2 doesn't have a default constructor
struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;
};
```

As we saw above, initializer lists are particularly useful when writing constructors for types like `Ray2`. Without them, we get a compiler error:

```
struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;

    Ray2(float originX, float originY, float directionX,
float directionY)
        // Compiler error
        // Origin and Direction don't have a default
constructor
        // The (float, float) constructor needs to be
called
        // That needs to be done here in the initializer
list
    {
        // Don't have Vector2 objects to initialize
```

```
        // They needed to be initialized in the
initializer list
        Origin.X = originX;
        Origin.Y = originY;
        Origin.X = directionX;
        Origin.Y = directionY;
    }
};
```

With initializer lists, we can call the non-default constructor to initialize these data members just before the constructor body runs:

```
struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;

    Ray2(float originX, float originY, float directionX,
float directionY)
        : Origin(originX, originY), Direction{directionX,
directionY}
    {
    }
};
```

Copy and Move Constructors

A copy constructor is a constructor that takes an lvalue reference to the same type of struct. This is typically a `const` reference. We'll cover `const` more later in the book, but for now it can be thought of as "read only."

Similarly, a move constructor takes an *rvalue* reference to the same type of struct. Here's all four of these in `Vector2`:

```
struct Vector2
{
    float X;
    float Y;

    // Default constructor
    Vector2()
    {
        X = 0;
        Y = 0;
    }

    // Copy constructor
    Vector2(const Vector2& other)
    {
        X = other.X;
        Y = other.Y;
    }
```

```
    // Copy constructor (parameter is not const)
    Vector2(Vector2& other)
    {
        X = other.X;
        Y = other.Y;
    }

    // Move constructor
    Vector2(const Vector2&& other)
    {
        X = other.X;
        Y = other.Y;
    }

    // Move constructor (parameter is not const)
    Vector2(Vector2&& other)
    {
        X = other.X;
        Y = other.Y;
    }
};
```

Unlike C#, the C++ compilers will generate a copy constructor if we don't define any copy or move constructors and all the data members can be copy-constructed. Likewise, the compiler will generate a move constructor if we don't define any copy or move constructors and all the data members can be move-constructed. So the compiler will generate both a copy and a move constructor for `Vector2` and `Ray2` here:

```
struct Vector2
{
    float X;
    float Y;

    // Compiler generates copy constructor:
    // Vector2(const Vector2& other)
    //     : X(other.X), Y(other.Y)
    // {
    // }

    // Compiler generates move constructor:
    // Vector2(const Vector2&& other)
    //     : X(other.X), Y(other.Y)
    // {
    // }
};

struct Ray2
{
    Vector2 Origin;
    Vector2 Direction;

    // Compiler generates copy constructor:
    // Ray2(const Ray2& other)
    //     : Origin(other.Origin),
Direction(other.Direction)
    // {
```

```
    // }

    // Compiler generates move constructor:
    // Ray2(const Ray2&& other)
    //     : Origin(other.Origin),
Direction(other.Direction)
    // {
    // }
};
```

The parameter to these compiler-generated copy and move constructors is `const` if there are `const` copy and move constructors available to call and non-`const` if there aren't.

As with default constructors, we can use `= default` to tell the compiler to generate copy and move constructors:

```
struct Vector2
{
    float X;
    float Y;

    // Inside struct
    Vector2(const Vector2& other) = default;
};

struct Ray2
{
    Vector2 Origin;
```

```
    Vector2 Direction;

    Ray2(Ray2&& other);
};

// Outside struct
// Explicitly defaulted move constructor can't take const
Ray2::Ray2(Ray2&& other) = default;
```

We can also use `=delete` to disable compiler-generated copy and move constructors:

```
struct Vector2
{
    float X;
    float Y;

    Vector2(const Vector2& other) = delete;
    Vector2(const Vector2&& other) = delete;
};
```

Destructors

C# classes can have finalizers, often called destructors. C# structs cannot, but C++ structs can.

Unlike constructors, which are pretty similar between the two languages, C++ destructors are extremely different. These differences have huge impacts on how C++ code is designed and written.

Syntactically, C++ destructors look the same as C# class finalizers/destructors: we just put a `~` before the struct name and take no parameters.

```
struct File
{
    FILE* handle;

    // Constructor
    File(const char* path)
    {
        // fopen() opens a file
        handle = fopen(path, "r");
    }

    // Destructor
    ~File()
    {
        // fclose() closes the file
        fclose(handle);
```

```
    }
};
```

We can also put the definition outside the struct:

```
struct File
{
    FILE* handle;

    // Constructor
    File(const char* path)
    {
        // fopen() opens a file
        handle = fopen(path, "r");
    }

    // Destructor declaration
    ~File();
};

// Destructor definition
File::~File()
{
    // fclose() closes the file
    fclose(handle);
}
```

The destructor is usually called implicitly, but it can be called explicitly:

```
File file("myfile.txt");
file.~File(); // Call destructor
```

The basic purpose of both C# finalizers and C++ destructors is the same: do some cleanup when the object goes away. In C#, an object "goes away" after it's garbage-collected. The timing of when the finalizer is called, if it is called at all, is highly complicated, non-deterministic, and multi-threaded.

In C++, an object's destructor is simply called when its lifetime ends:

```
void OpenCloseFile()
{
    File file("myfile.txt");
    DebugLog("file opened");
    // Compiler generates: file.~File();
}
```

This is ironclad. The language guarantees that the destructor gets called no matter what. Consider an exception, which we'll cover in more depth later in the book but acts similarly to C# exceptions:

```
void OpenCloseFile()
{
    File file("myfile.txt");
    if (file.handle == nullptr)
```

```
    {
        DebugLog("file filed to open");
        // Compiler generates: file.~File();
        throw IOException();
    }
    DebugLog("file opened");
    // Compiler generates: file.~File();
}
```

No matter how `file` goes out of scope, its destructor is called first.

Even a `goto` based on runtime computation can't get around the destructor:

```
void Foo()
{
    label:
    File file("myfile.txt");
    if (RollRandomNumber() == 3)
    {
        // Compiler generates: file.~File();
        return;
    }
    shouldReturn = true;
    // Compiler generates: file.~File();
    goto label;
}
```

To briefly see how this impacts the design of C++ code, let's add a `GetSize` member function to `File` so it can do something useful. Let's also add some exception-based error handling:

```
struct File
{
    FILE* handle;

    File(const char* path)
    {
        handle = fopen(path, "r");
        if (handle == nullptr)
        {
            throw IOException();
        }
    }

    long GetSize()
    {
        long oldPos = ftell(handle);
        if (oldPos == -1)
        {
            throw IOException();
        }

        int fseekRet = fseek(handle, 0, SEEK_END);
        if (fseekRet != 0)
        {
            throw IOException();
```

```
        }

        long size = ftell(handle);
        if (size == -1)
        {
            throw IOException();
        }

        fseekRet = fseek(handle, oldPos, SEEK_SET);
        if (fseekRet != 0)
        {
            throw IOException();
        }

        return size;
    }

    ~File()
    {
        fclose(handle);
    }
};
```

We can use this to get the size of the file like so:

```
long GetTotalSize()
{
    File fileA("myfileA.txt");
```

```
    File fileB("myfileB.txt");
    long sizeA = fileA.GetSize();
    long sizeB = fileA.GetSize();
    long totalSize = sizeA + sizeB;
    return totalSize;
}
```

The compiler generates several destructor calls for this. To see them all, let's see a pseudo-code version of what the constructor generates:

```
long GetTotalSize()
{
    File fileA("myfileA.txt");

    try
    {
        File fileB("myfileB.txt");
        try
        {
            long sizeA = fileA.GetSize();
            long sizeB = fileA.GetSize();
            long totalSize = sizeA + sizeB;
            fileB.~File();
            fileA.~File();
            return totalSize;
        }
        catch (...) // Catch all types of exceptions
```

```
        {
            fileB.~File();
            throw; // Re-throw the exception to the outer
catch
        }
    }
    catch (...) // Catch all types of exceptions
    {
        fileA.~File();
        throw; // Re-throw the exception
    }
}
```

In this expanded view, we see that the compiler generates destructor calls in every possible place where `fileA` or `fileB` could end their lifetimes. It's impossible for us to forget to call the destructor because the compiler thoroughly adds all the destructor calls for us. We know by design that neither file handle will ever leak.

Another aspect of destructors is also visible here: they're called on objects in the reverse order that the constructors are called. Because we declared `fileA` first and `fileB` second, the constructor order is `fileA` then `fileB` and the destructor order is `fileB` then `fileA`.

The same ordering goes for the data members of a struct:

```
struct TwoFiles
{
    File FileA;
    File FileB;
```

```
};

void Foo()
{
    // If we write this code...
    TwoFiles tf;

    // The compiler generates constructor calls: A then B
    // Pseudo-code: can't really call a constructor
directly
    tf.FileA();
    tf.FileB();

    // Then destructor calls: B then A
    tf.~FileB();
    tf.~FileA();
}
```

This explains why we can't change the order of data members in an initializer list: the compiler needs to be able to generate the reverse order of destructor calls no matter what the constructor does.

Finally, the compiler generates a destructor implicitly:

```
struct TwoFiles
{
    File FileA;
    File FileB;
```

```
    // Compiler-generated destructor
    ~TwoFiles()
    {
        FileB.~File();
        FileA.~File();
    }
};
```

We can use `= default` to explicitly tell it to do this:

```
// Inside the struct
struct TwoFiles
{
    File FileA;
    File FileB;

    ~TwoFiles() = default;
};

// Outside the struct
struct TwoFiles
{
    File FileA;
    File FileB;

    ~TwoFiles();
};
TwoFiles::~TwoFiles() = default;
```

And we can stop the compiler from generating one with `= delete`:

```
struct TwoFiles
{
    File FileA;
    File FileB;

    ~TwoFiles() = delete;
};
```

The compiler generates a destructor as long as we haven't written one and all of the data members can be destructed.

Conclusion

At their most basic, constructors are the same in C# and C++. The two languages quickly depart though with implicitly or explicitly compiler-generated default, copy, and move constructors, support for writing custom default constructors, strict initialization ordering, and initializer lists.

Destructors are starkly different from C# finalizers/destructors. They're called predictibly as soon as the object's lifetime ends, rather than on another thread long after the object is released or perhaps never called at all. The paradigm is similar to C#'s `using (IDisposable)`, but there's no need to add the `using` part and no way to forget it. They also strictly order destruction in the reverse of construction and provide us the option to generate or not generate destructors for us.

13. Initialization

Explicit Constructors

Before we get to initialization, we need to talk a little more about how struct objects are created. First, all of the constructors we write may be optionally declared as `explicit`:

```
struct Vector2
{
    float X;
    float Y;

    explicit Vector2(const Vector2& other)
    {
        X = other.X;
        Y = other.Y;
    }
};
```

In C++20, this can be conditional on a compile-time constant expression put into parentheses after the `explicit` keyword:

```
struct Vector2
{
    float X;
    float Y;
```

```
    explicit (2 > 1) Vector2(const Vector2& other)
    {
        X = other.X;
        Y = other.Y;
    }
};
```

When a constructor is `explicit`, it's no longer considered a "converting constructor." As we'll see below, some forms of initialization will no longer allow the constructor to be called implicitly.

User-Defined Conversion Operators

As with C#, we can write our own conversion operators from a struct to any other type:

```
struct Vector2
{
    float X;
    float Y;

    operator bool()
    {
        return X != 0 || Y != 0;
    }
};
```

Also as in C#, these can be `explicit`.

```
struct Vector2
{
    float X;
    float Y;

    explicit operator bool()
    {
        return X != 0 || Y != 0;
```

```
    }
};
```

They can also be conditionally `explicit` as of C++20:

```
struct Vector2
{
    float X;
    float Y;

    explicit (2 > 1) operator bool()
    {
        return X != 0 || Y != 0;
    }
};
```

There's no `implicit` keyword like we have in C#. To make one `implicit`, just don't add `explicit`.

In C#, user-defined conversion operators are `static` and take a parameter of the same type as the struct they're defined in. In C++, they're non-static and `this` is used implicitly or explicitly instead of the parameter.

Like other overloaded operators, they may be called explicitly. It's rare to see this, but it's allowed:

```
Vector2 v1;
v1.X = 2;
```

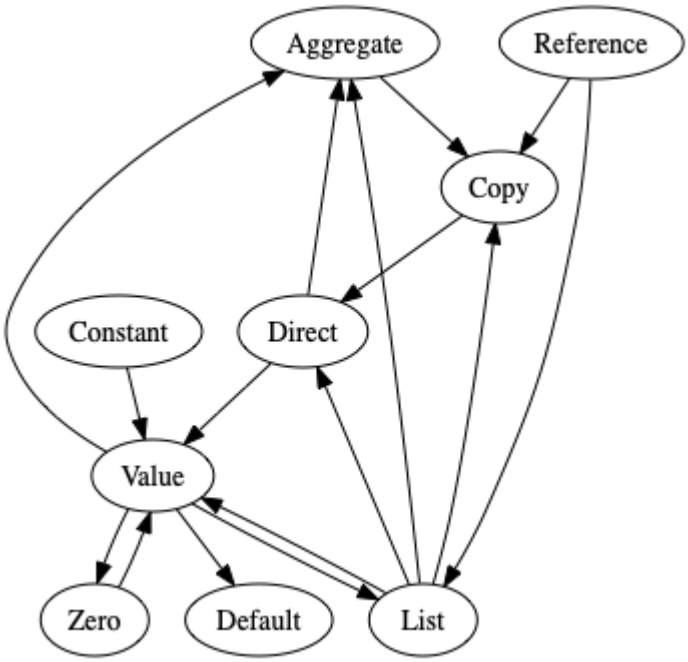
```
v1.Y = 4;
bool b = v1.operator bool();
```

Initialization Types

C++ classifies initialization into the following types:

- Default
- Aggregate
- Constant
- Copy
- Direct
- List
- Reference
- Value
- Zero

The rules for how a type works frequently defers to the rules for how another type works. This is similar to a function calling another function. It creates a dependency of one type on another type. These dependencies frequently form cycles in the graph, which looks roughly like this:



This means that as we go through the initialization types we're going to refer to other initialization types that we haven't seen yet. Feel free to jump ahead to the referenced type or come back to revisit a type after reading about its references later on in the chapter.

As for terminology, we often say that a variable is "X-initialized" to mean that it is initialized using the rules of the "X" initialization type. For example, "MyVar is direct-initialized" means "MyVar is initialized according to the rules of the direct initialization type."

Default Initialization

Default initialization happens when a variable is declared with no initializer:

```
T object;
```

It also happens when calling a constructor that doesn't mention a data member:

```
struct HasDataMember
{
    T Object;
    int X;

    HasDataMember()
        : X(123) // No mention of Object
    {
    }
};
```

If the type (`T`) is a struct, its default constructor is called. If it's an array, every element of the array is default-initialized:

```
struct ConstructorLogs
{
    ConstructorLogs()
```

```
    {
        DebugLog("default");
    }
};

ConstructorLogs single; // Prints "default"
ConstructorLogs array[3]; // Prints "default", "default",
"default"
```

For all other types, nothing happens. This includes primitives, enums, and pointers. Using one of these objects is undefined behavior and may cause severe errors since the compiler can generate any code it wants to.

```
float f;
DebugLog(f); // Undefined behavior!
```

Default initialization isn't allowed for these kinds of variables if they're `const` since there would be no way to initialize them later:

```
void Foo()
{
    const float f; // Compiler error: default initializer
does nothing
}
```

One exception is for static variables, including both `static` data members of structs and global variables. These are zero-initialized:

```
const float f; // OK: this is a global variable

struct HasStatic
{
    static float X;
};
float HasStatic::X; // OK: this is a static data member
```

It is also allowed if there's a default constructor to call because that initializes the variable:

```
const HasDataMember single; // OK: calls default
constructor

struct NoDefaultConstructor
{
    NoDefaultConstructor() = delete;
};

const NoDefaultConstructor ndc; // Compiler error: no
default constructor
```

References, both lvalue and rvalue, are never default-initialized. They have their own initialization type which we'll cover below: reference initialization.

Copy Initialization

Copy initialization has several forms:

```
// Assignment style
T object = other;

// Function call
func(other)

// Return value
return other;

// Array assigned to curly braces
T array[N] = {other};
```

For the first three forms, only one object is involved. That object's copy constructor is called with `other` being passed in as the argument:

```
struct Logs
{
    Logs() = default;

    Logs(const Logs& logs)
    {
        DebugLog("copy");
    }
```

```
};

Logs Foo(Logs a)
{
    Logs b = a; // "copy" for assignment style
    return a; // "copy" for return value
}

Logs x;
Foo(x); // "copy" for function call
```

This is no longer allowed if the copy constructor is `explicit`:

```
struct Logs
{
    Logs() = default;

    explicit Logs(const Logs& logs)
    {
        DebugLog("copy");
    }
};

Logs Foo(Logs a)
{
    Logs b = a; // Compiler error: copy constructor is
explicit
    return a; // Compiler error: copy constructor is
```

```
explicit
}

Logs x;
Foo(x); // Compiler error: copy constructor is explicit
```

User-defined conversion operators can also be called by the same three forms of copy initialization:

```
struct ConvertLogs
{
    ConvertLogs() = default;

    operator bool()
    {
        DebugLog("convert");
        return true;
    }
};

bool Foo(bool b)
{
    ConvertLogs x;
    return x; // "convert" for return value
}

ConvertLogs x;
bool b = x; // "convert" for assignment style
```

```
Foo(x); // "convert" for function call
```

The return value of the user-defined conversion operator, a `bool` in this example, is then used to direct-initialize the variable.

As with the copy constructor, making the user-defined conversion operator `explicit` disables copy initialization and makes all of these “convert” lines generate compiler errors just like when we made the copy constructor `explicit`.

For non-struct types like primitives, enums, and pointers, the value is simply copied:

```
int x = y;
```

The last form deals with arrays. This happens during aggregate initialization.

Aggregate Initialization

Aggregate initialization has the following forms:

```
// Assign curly braces
T object = { val1, val2 };

// No-assign curly braces
T object{ val1, val2 };

// Assign curly braces with "designators" (data member
names)
T object = { .designator=val1, .designator=val2 };

// No-assign curly braces with "designators" (data member
names)
T object{ .designator=val1, .designator=val2 };

// Parentheses
T object(val1, val2);
```

All of these forms work on types (`T`) that are considered "aggregates." That includes arrays and structs that don't have any constructors except those using `= default`.

The elements of these arrays and data members of these structs are copy-initialized with the given values: `val1`, `val2`, etc. This is done in index order starting at the first element for arrays. With structs, this is

done in the order that data members are declared, just like a constructor's initializer list does.

Designators are available as of C++20. They're similar to C#'s "object initializers": `Vector2 vec = {X=2, Y=4};`. They must be in the same order as the struct's data members and all values must have designators.

```
struct Vector2
{
    float X;
    float Y;
};

Vector2 v1 = { 2, 4 };
DebugLog(v1.X, v1.Y); // 2, 4

Vector2 v2{2, 4};
DebugLog(v2.X, v2.Y); // 2, 4

Vector2 v3 = { .X=2, .Y=4 };
DebugLog(v3.X, v3.Y); // 2, 4

Vector2 v4{ .X=2, .Y=4 };
DebugLog(v4.X, v4.Y); // 2, 4

Vector2 v5(2, 4);
DebugLog(v5.X, v5.Y); // 2, 4
```

It's a compiler error to pass more values than there are data members or array elements:

```
Vector2 v5 = {2, 4, 6}; // Compiler error: too many data members
float a1[2] = {2, 4, 6}; // Compiler error: too many array elements
```

We can, however, pass fewer values than data members or array elements. The remaining data members are initialized with their default member initializers. If they don't have default member initializers, they're copy-initialized from an empty list (`{}`).

```
struct DefaultedVector2
{
    float X = 1;
    float Y;
};

DefaultedVector2 dv1 = {2};
DebugLog(dv1.X, dv1.Y); // 2, 0

float a2[2] = {2};
DebugLog(a2[0], a2[1]); // 2, 0
```

If a data member is an lvalue or rvalue reference, not passing it is a compiler error because it could never be initialized later on due to how references work.

```
struct HasRef
{
    int X;
    int& R;
};

HasRef hr = {123}; // Compiler error: reference data
member not initialized
```

There are special rules for aggregate-initializing arrays from string literals:

```
// a1 has length 4 and contains: 'a', 'b', 'c', 0
char a1[4] = "abc";

// Length is optional. This is the same as a1.
char a2[] = "abc";

// Curly braces are optional. This is the same as a1.
char a3[] = {"abc"};

// Compiler error: array too small to fit the string
literal's contents
char a4[1] = "abc";

// Extra array elements are zero-initialized
```

```
// a5 has length 6 and contains: 'a', 'b', 'c', 0, 0, 0
char a5[6] = "abc";
```

List Initialization

There are two sub-types of list initialization. First, “direct list initialization” has these forms:

```
// Named variable
T object{val1, val2};

// Unnamed temporary variable
T{val1, val2}

struct MyStruct
{
    // Data member
    T member{val1, val2};
};

MyStruct::MyStruct()
    // Initializer list entry
    : member{val1, val2}
{
}
```

Second, there’s “copy list initialization” with these forms:

```
// Named variable
T object = {val1, val2};
```

```
// Function call
func({val1, val2})

// Return value
return {val1, val2};

// Overloaded subscript operator call
object[{val1, val2}]

// Assignment
object = {val1, val2}

struct MyStruct
{
    // Data member
    T member = {val1, val2};
};
```

The compiler chooses what to do by essentially using a pretty long series of `if`–`else` decisions.

First, if there's a single value of the same type then it copy-initializes for copy list initialization and direct-initializes for direct list initialization:

```
Vector2 vec;
vec.X = 2;
vec.Y = 4;
```

```
// Direct list initialization direct-initializes vecA
with vec
Vector2 vecA{vec};
DebugLog(vecA.X, vecA.Y); // 2, 4

// Copy list initialization copy-initializes vecB with
vec
Vector2 vecB = {vec};
DebugLog(vecB.X, vecB.Y); // 2, 4
```

Second, if the variable is a character array and there's a single value of the same character type then the variable is aggregate-initialized:

```
char array[1] = {'x'}; // Aggregate-initialized
DebugLog(array[0]); // x
```

Third, if the variable to initialize is an aggregate type then it's aggregate-initialized:

```
Vector2 vec = {2, 4}; // Aggregate-initialized
DebugLog(vec.X, vec.Y); // 2, 4
```

Fourth, if no values are passed in the curly braces and the variable to initialize is a struct with a default constructor then it's value-initialized:

```
struct NonAggregateVec2
{
```

```
    float X;
    float Y;

    NonAggregateVec2()
    {
        X = 2;
        Y = 4;
    }
};

NonAggregateVec2 vec = {}; // Value-initialized
DebugLog(vec.X, vec.Y); // 2, 4
```

Fifth, if the variable has a constructor that takes only the Standard Library's `std::initializer_list` type then that constructor is called. We haven't covered any of the Standard Library yet, but the details of this type aren't really important at this point. Suffice to say that this is the C++ equivalent to initializing collections in C#: `List<int> list = new List<int> { 2, 4 };`.

```
struct InitListVec2
{
    float X;
    float Y;

    InitListVec2(std::initializer_list<float> vals)
    {
        X = *vals.begin();
        Y = *(vals.begin() + 1);
```

```
    }
};

InitListVec2 vec = {2, 4};
DebugLog(vec.X, vec.Y); // 2, 4
```

Sixth, if any constructor matches the passed values then the one that matches best is called:

```
struct MultiConstructorVec2
{
    float X;
    float Y;

    MultiConstructorVec2(float x, float y)
    {
        X = x;
        Y = y;
    }

    MultiConstructorVec2(double x, double y)
    {
        X = x;
        Y = y;
    }
};

MultiConstructorVec2 vec1 = {2.0f, 4.0f}; // Call (float,
```

```
float) version
DebugLog(vec1.X, vec1.Y); // 2, 4

MultiConstructorVec2 vec2 = {2.0, 4.0}; // Call (double,
double) version
DebugLog(vec2.X, vec2.Y); // 2, 4
```

Seventh, if the variable is a (scoped or unscoped) enumeration and a single value of that type is passed with direct list initialization, the variable is initialized with that value:

```
enum struct Color : uint32_t
{
    Blue = 0x0000ff
};

Color c = {Color::Blue};
DebugLog(c); // 255
```

Eighth, if the variable isn't a struct, only one value is passed, and that value isn't a reference, then the variable is direct-initialized:

```
float f = {3.14f};
DebugLog(f); // 3.14
```

Ninth, if the variable isn't a struct, the curly braces have only one value, and the variable isn't a reference or is a reference to the type of the single value, then the variable is direct-initialized for direct list

initialization or copy-initialized for copy list initialization with the value:

```
float f = 3.14f;

float& r1{f}; // Direct list initialization direct-initializes
DebugLog(r1); // 3.14

float& r2 = {f}; // Copy list initialization copy-initializes
DebugLog(r2); // 3.14

float r3{f}; // Direct list initialization direct-initializes
DebugLog(r3); // 3.14

float r4 = {f}; // Copy list initialization copy-initializes
DebugLog(r4); // 3.14
```

Tenth, if the variable is a reference to a different type than the one value passed then a temporary reference to the value's type is created, list-initialized, and bound to the variable. The variable must be `const` for this to work:

```
float f = 3.14;
```

```
const int32_t& r1 = f;
DebugLog(r1); // 3

int32_t& r2 = f; // Compiler error: not const
DebugLog(r2);
```

Eleventh, and lastly, if no values are passed then the variable is value-initialized:

```
float f = {};
DebugLog(f); // 0
```

One final detail to note is that the values passed in the curly braces are evaluated in order. This is unlike the arguments passed to a function which are evalutated in an order determined by the compiler.

Reference Initialization

As mentioned above, references have their own type of initialization. Here are the forms it takes:

```
// lvalue reference variables
T& ref = object;
T& ref = {val1, val2};
T& ref(object);
T& ref{val1, val2};

// rvalue reference variables
T&& ref = object;
T&& ref = {val1, val2};
T&& ref(object);
T&& ref{val1, val2};

// Function calls
/* Assume */ void func(T& val); /* or */ void func(T&&
val);
func(object)
func({val1, val2})

// Return values
T& func() { T t; return t; }
T&& func() { return T(); }

// Constructor initializer lists
```

```
MyStruct::MyStruct()
    : lvalueRef(object)
    , rvalueRef(object)
{
}
```

If curly braces are provided, the reference is list-initialized:

```
float&& f = {3.14f};
DebugLog(f); // 3.14
```

Otherwise, the reference follows reference initialization rules. These are effectively another series of `if`–`else` decisions, but a much shorter series than with list initialization.

First, for lvalue references of the same type the reference simply binds to the passed object:

```
float f = 3.14f;
float& r = f;
DebugLog(r); // 3.14
```

When the variable is an lvalue reference but it has a different type than the passed object, if there's a user-defined conversion function then it's called and the variable is bound to the return value:

```
float pi = 3.14f;
```

```
struct ConvertsToPi
{
    operator float&()
    {
        return pi;
    }
};

ConvertsToPi ctp;
float& r = ctp; // User-defined conversion operator
called
DebugLog(r); // 3.14
```

In all other cases the passed expression is evaluated into a temporary variable and the reference is bound to that:

```
float Add(float a, float b)
{
    return a + b;
}

// Call function, store return value in temporary, bind
reference to temporary
float&& sum = Add(2, 4);
DebugLog(sum); // 6
```

Temporary variables created by reference initialization have their lifetimes extended to match the lifetime of the reference. There are a

few exceptions. First, returned references are always “dangling” as what they refer to ends its lifetime when the function exits. Second, and similarly, references to function arguments also end their lifetime when the function exits.

```
float&& Dangling1()
{
    return 3.14f; // Returned temporary ends its lifetime
here
}

float& Dangling2(float x)
{
    return x; // Returned argument ends its lifetime here
}

DebugLog(Dangling1()); // Undefined behavior
DebugLog(Dangling2(3.14f)); // Undefined behavior
```

Third, the reference data members or elements of an aggregate only have their lifetime extended when curly braces, not parentheses, are used:

```
struct HasRvalueRef
{
    float&& Ref;
};

// Curly braces used. Lifetime of float with 3.14f value
```

```
extended.
HasRvalueRef hrr1{3.14f};
DebugLog(hrr1.Ref); // 3.14

// Parentheses used. Lifetime of float with 3.14f value
NOT extended.
HasRvalueRef hrr2(3.14f);
DebugLog(hrr2.Ref); // Undefined behavior. Ref has ended
its lifetime.
```

Value Initialization

Value initialization can look like this:

```
// Variable
T object{};

// Temporary variable (i.e. it has no name)
T()
T{}

// Initialize a data member in an initializer list
MyStruct::MyStruct()
    : member1() // Parentheses version
    , member2{} // Curly braces version
{
}
```

Value initialization always defers to another type of initialization. Here's how it decides which type to use:

If curly braces are used and the variable is an aggregate, it's aggregate-initialized.

```
Vector2 vec{2, 4}; // Aggregate initialization
DebugLog(vec.X, vec.Y); // 2, 4
```

If the variable is a struct that doesn't have a default constructor but it does have a constructor that takes only a `std::initializer_list`, the variable is list-initialized with an empty list (i.e. `{}`).

```
struct InitListVec2
{
    float X;
    float Y;

    InitListVec2(std::initializer_list<float> vals)
    {
        int index = 0;
        float x = 0;
        float y = 0;
        for (float cur : vals)
        {
            switch (index)
            {
                case 0: x = cur; break;
                case 1: y = cur; break;
            }
        }
        X = x;
        Y = y;
    }
};

InitListVec2 vec{}; // List initialization (passes empty
```

```
list)
DebugLog(vec.X, vec.Y); // 0, 0
```

If the variable is a struct with no default constructor, it's default-initialized.

```
struct Vector2
{
    float X;
    float Y;

    Vector2() = delete;
};

Vector2 vec{}; // Default-initialized
DebugLog(vec.X, vec.Y); // 0, 0
```

If the default constructor was generated by the compiler, the variable is zero-initialized *then* direct-initialized if any of the data members have default initializers (i.e. `float X = 0;`).

```
struct Vector2
{
    float X = 2;
    float Y = 4;
};

Vector2 vec{}; // Zero initialization then direct
```

```
initialization
DebugLog(vec.X, vec.Y); // 2, 4
```

If the variable is an array, each element is value-initialized.

```
float arr[2]{}; // Elements value-initialized
DebugLog(arr[0], arr[1]); // 0, 0
```

If none of the above apply, the variable is zero-initialized.

```
float x{}; // Zero-initialized
DebugLog(x); // 0
```

Direct initialization

Here are the forms direct initialization can take:

```
// Parentheses with single value
T object(val);

// Parentheses with multiple values
T object(val1, val2);

// Curly braces with single value
T object{val};

MyStruct::MyStruct()
    // Parentheses in initializer list
    : member(val1, val2)
{
}
```

All of these look for a constructor matching the passed values. If one is found, the one that matches best is called to initialize the variable.

```
struct MultiConstructorVec2
{
    float X;
    float Y;

    MultiConstructorVec2(float x, float y)
```

```
    {
        X = x;
        Y = y;
    }

    MultiConstructorVec2(double x, double y)
    {
        X = x;
        Y = y;
    }
};

MultiConstructorVec2 vec1{2.0f, 4.0f}; // Call (float, float) version
DebugLog(vec1.X, vec1.Y); // 2, 4

MultiConstructorVec2 vec2{2.0, 4.0}; // Call (double, double) version
DebugLog(vec2.X, vec2.Y); // 2, 4
```

If no constructor matches or the variable isn't a struct but it is an aggregate, the variable is aggregate-initialized.

```
struct Vector2
{
    float X;
    float Y;
};
```

```
// No constructor matches, but Vector2 is an aggregate
Vector2 vec{2, 4}; // Aggregate initialization
DebugLog(vec.X, vec.Y); // 2, 4
```

As of C++20, the variable can be an array. In this case the rules of aggregate initialization apply. For example, passing too many values is a compiler error.

```
float a1[2]{2, 4}; // Aggregate initialization
DebugLog(a1[0], a1[1]); // 2, 4

float a2[2]{2, 4, 6, 8}; // Compiler error: too many values
```

There's one type-specific exception to this. If the variable is a `bool` and the value is `nullptr`, the variable becomes `false`.

```
bool b{nullptr};
DebugLog(b); // false
```

One common mistake with the parentheses forms of direct initialization is to create ambiguity between initialization of a variable and a function declaration. Consider this code:

```
struct Enemy
{
    float X;
```

```
    float Y;
};

struct Vector2
{
    float X;
    float Y;

    Vector2() = default;

    Vector2(Enemy enemy)
    {
        X = enemy.X;
        Y = enemy.Y;
    }
};

Vector2 defaultEnemySpawnPoint(Enemy());
```

The last line is ambiguous. The naming makes us think it's a variable with type `Vector2` named `defaultEnemySpawnPoint` that's being direct-initialized with a value-initialized temporary `Enemy` variable.

Another way to read that line is that it declares a function named `defaultEnemySpawnPoint` that returns a `Vector2` and takes an unnamed pointer to a function that takes no parameters and returns an `Enemy`. In that alternate reading, we could write code like this:

```
// Definition of a function that satisfies the function
pointer type
Enemy cb()
{
    return {};
}

// Definition of the above declaration, intentional or
not
Vector2 defaultEnemySpawnPoint(Enemy())
{
    return {};
}

// It can be called with 'cb' as the function pointer
argument
defaultEnemySpawnPoint(cb);
```

The compiler always chooses the function declaration when this ambiguity arises. That means the above code is valid and actually works, but we'll get errors if we try to use `defaultEnemySpawnPoint` like a variable when it's actually a function:

```
// Compiler error: defaultEnemySpawnPoint is a function
// Functions have no X or Y data members to get
DebugLog(defaultEnemySpawnPoint.X,
defaultEnemySpawnPoint.Y);
```

Thankfully, it's easy to resolve the ambiguity by simply using the curly braces form of direct-initialization because the function pointer syntax doesn't use curly braces:

```
Vector2 defaultEnemySpawnPoint(Enemy{});
DebugLog(defaultEnemySpawnPoint.X,
defaultEnemySpawnPoint.Y); // 0, 0
```

Constant Initialization

Constant initialization has just two forms:

```
T& ref = constantExpression;
T object = constantExpression;
```

Both of these only apply when the variable is both `const` and static, such as for global variables and `static` struct data members. Otherwise, the variable is zero-initialized.

```
struct Player
{
    static const int32_t MaxHealth;

    int32_t Health;
};

// Constant-initialize a data member
const int32_t Player::MaxHealth = 100;

// Constant-initialize a global reference
const int32_t& defaultHealth = Player::MaxHealth;
```

This initialization happens before all other initialization, so it's safe to read from these variables during other kinds of initialization. That's even the case if that other initialization appears before the constant initialization:

```
struct Player
{
    static const int32_t MaxHealth;

    int32_t Health;
};

// 2) Aggregate initialization
Player localPlayer{Player::MaxHealth};

// 1) Constant initialization
const int32_t Player::MaxHealth = 100;
const int32_t& defaultHealth = Player::MaxHealth;

// 3) Normal code, not initialization
DebugLog(localPlayer.Health); // 100
```

Zero Initialization

Lastly, we have zero initialization. Unlike all the other types, it doesn't have any explicit forms. Instead, as we've seen above, other types of initialization may result in zero initialization:

```
// Static variable that's not constant-initialized
// Zero initialization still happens before other types
of initialization
static T object;

// During value initialization for non-struct types
// Includes struct data members and array elements
T();
T t = {};
T{};

// When initializing an array from a string literal
that's too short
// Remaining elements are zero-initialized
char array[N] = "";
```

Zero initialization sets primitives and all padding bits of structs to `0`. It doesn't do anything to references.

Conclusion

As we've now seen, initialization is a far more complex topic in C++ than it is in C#. The main reason is that C++ provides far more features. Supporting default constructors, temporary variables, arrays, references, function pointers, `const`, string literals, and so forth requires a fair amount more syntax.

14. Inheritance

Base Structs

The basic syntax for inheritance, also called derivation, looks the same in C++ for structs as it does in C# for classes. We just add `: BaseType` after the name of the struct:

```
struct GameEntity
{
    static const int32_t MaxHealth = 100;
    int32_t Health = MaxHealth;

    float GetHealthPercent()
    {
        return ((float)Health) / MaxHealth;
    }
};

struct MovableGameEntity : GameEntity
{
    float Speed = 0;
};
```

When *declaring*, not *defining*, a struct that inherits from another struct, we omit the base struct name:

```
struct MovableGameEntity; // No base struct name

struct MovableGameEntity : GameEntity
{
    float Speed = 0;
};
```

The meaning of this inheritance is the same in C++ as in C#: `MovableGameEntity` "is a" `GameEntity`. That means all the data members and member functions of `GameEntity` are made part of `MovableGameEntity` as a sub-object. We can write code to use the contents of both the parent and the child struct:

```
MovableGameEntity mge{};
mge.Health = 50;
DebugLog(mge.Health, mge.Speed, mge.GetHealthPercent());
// 50, 0, 0.5
```

Normally, any object must have a size of at least one byte. One exception to this rule is when a base struct has no non-`static` data members. In that case, it may add zero bytes to the size of the structs that derive from it. An exception to this exception is if the first non-`static` data member is also the base struct type.

```
// Has no non-static data members
struct Empty
{
    void SayHello()
```

```
    {
        DebugLog("hello");
    }
};

// Size not increased by deriving from Empty
struct Vector2 : Empty
{
    float X;
    float Y;
};

// Size increased because first non-static data member is
also an Empty
struct ExceptionToException : Empty
{
    Empty E;
    int32_t X;
};

void Foo()
{
    Vector2 vec{};
    DebugLog(sizeof(vec)); // 8 (size not increased)
    vec.SayHello(); // "hello"

    DebugLog(sizeof(ExceptionToException)); // 8 (size
```

```
increased from 4!)
}
```

The “is a” relationship continues in that pointers and references to a `MovableGameEntity` are compatible with pointers and references to an `GameEntity`:

```
MovableGameEntity mge{};
GameEntity* pge = &mge; // Pointers are compatible
GameEntity& rge = mge; // References are compatible
DebugLog(mge.Health, pge->Health, rge.Health); // 100,
100, 100
```

Derived structs can have members with the same names as their base structs. For example, `ArmoredGameEntity` might get extra health from its armor:

```
struct ArmoredGameEntity : GameEntity
{
    static const int32_t MaxHealth = 100;
    int32_t Health = MaxHealth;
};
```

This introduces ambiguity when referring to the `Health` member of an `ArmoredGameEntity`: is it the data member declared in `GameEntity` or `ArmoredGameEntity`? The compiler chooses the member of the type being referred to:

```
// Referring to ArmoredGameEntity, so use Health in
ArmoredGameEntity
ArmoredGameEntity age{};
DebugLog(age.Health); // 50

// Referring to GameEntity, so use Health in GameEntity
GameEntity& ge = age;
DebugLog(ge->Health); // 100
```

To resolve this ambiguity, we can explicitly refer to the members of a particular sub-object in the inheritance hierarchy using the scope resolution operator: `StructType::Member`.

```
struct ArmoredGameEntity : GameEntity
{
    static const int32_t MaxHealth = 50;
    int32_t Health = MaxHealth;

    void Die()
    {
        Health = 0; // Health in ArmoredGameEntity
        GameEntity::Health = 0; // Health in GameEntity
    }
};

ArmoredGameEntity age{};
GameEntity& ge = age;
DebugLog(age.Health, ge.Health); // 50, 100
```

```
age.Die();
DebugLog(age.Health, ge.Health); // 0, 0
```

Although it's uncommon to do so, we can also refer to specific structs from outside the struct hierarchy:

```
ArmoredGameEntity age{};
ArmoredGameEntity* page = &age;

// Refer to Health in GameEntity with
age.GameEntity::Health
DebugLog(age.Health, age.GameEntity::Health); // 50, 100

age.Die();

// Refer to Health in GameEntity via a pointer with page-
>GameEntity::Health
DebugLog(age.Health, page->GameEntity::Health); // 0, 0
```

This is sort of like using `base` in C#, except that we can refer to *any* struct in the hierarchy rather than just the immedate base class type.

```
struct MagicArmoredGameEntity : ArmoredGameEntity
{
    static const int32_t MaxHealth = 20;
    int32_t Health = MaxHealth;
};
```

```
MagicArmoredGameEntity mage{};
DebugLog(mage.Health); // 20
DebugLog(mage.ArmoredGameEntity::Health); // 50
DebugLog(mage.GameEntity::Health); // 100
```

Constructors and Destructors

As in C#, constructors are called from the top of the hierarchy to the bottom. Unlike C#'s non-deterministic destructors/finalizers, C++ destructors are called in the same order as constructors. Recall that data members are constructed in declaration order and destructed in the reverse order. These two properties combine to give the following deterministic order:

```
struct LogLifecycle
{
    const char* str;

    LogLifecycle(const char* str)
        : str(str)
    {
        DebugLog(str, "constructor");
    }

    ~LogLifecycle()
    {
        DebugLog(str, "destructor");
    }
};

struct GameEntity
{
    LogLifecycle a{"GameEntity::a"};
    LogLifecycle b{"GameEntity::b"};
```

```
    GameEntity()
    {
        DebugLog("GameEntity constructor");
    }

    ~GameEntity()
    {
        DebugLog("GameEntity destructor");
    }
};

struct ArmoredGameEntity : GameEntity
{
    LogLifecycle a{"ArmoredGameEntity::a"};
    LogLifecycle b{"ArmoredGameEntity::b"};

    ArmoredGameEntity()
    {
        DebugLog("ArmoredGameEntity constructor");
    }

    ~ArmoredGameEntity()
    {
        DebugLog("ArmoredGameEntity destructor");
    }
};
```

```
void Foo()
{
    ArmoredGameEntity age{};
    DebugLog("--after variable declaration--");
} // Note: destructor of 'age' called here

// Logs printed:
//   GameEntity::a, constructor
//   GameEntity::b, constructor
//   GameEntity constructor
//   ArmoredGameEntity::a, constructor
//   ArmoredGameEntity::b, constructor
//   ArmoredGameEntity constructor
//   --after variable declaration--
//   ArmoredGameEntity destructor
//   ArmoredGameEntity::b, destructor
//   ArmoredGameEntity::a, destructor
//   GameEntity destructor
//   GameEntity::b, destructor
//   GameEntity::a, destructor
```

As we saw above, the default constructor of base structs is called implicitly. However, if there's no default constructor then it must be called explicitly. The syntax looks like it does in C# except the base struct type is used instead of the keyword `base`:

```
struct GameEntity
{
```

```
    static const int32_t MaxHealth = 100;
    int32_t Health;

    GameEntity(int32_t health)
    {
        Health = health;
    }
};

struct ArmoredGameEntity : GameEntity
{
    static const int32_t MaxArmor = 100;
    int32_t Armor = 0;

    ArmoredGameEntity()
        : GameEntity(MaxHealth)
    {
    }
};
```

This is actually just part of the initializer list that we use to initialize data members:

```
struct ArmoredGameEntity : GameEntity
{
    static const int32_t MaxArmor = 100;
    int32_t Armor = 0;
```

```
    ArmoredGameEntity()
        : GameEntity(MaxHealth), Armor(MaxArmor)
    {
    }
};
```

Since the order of initialization is always as above, it doesn't matter what order we put the initializers in. This is true for both data members and base structs:

```
struct ArmoredGameEntity : GameEntity
{
    static const int32_t MaxArmor = 100;
    int32_t Armor = 0;

    ArmoredGameEntity()
        // Order doesn't matter
        // GameEntity is still initialized before Armor
        : Armor(MaxArmor), GameEntity(MaxHealth)
    {
    }
};
```

Multiple Inheritance

Unlike C#, C++ supports structs that derive from multiple structs:

```
struct HasHealth
{
    static const int32_t MaxHealth = 100;
    int32_t Health = MaxHealth;
};

struct HasArmor
{
    static const int32_t MaxArmor = 20;
    int32_t Armor = MaxArmor;
};

// Player derives from both HasHealth and HasArmor
struct Player : HasHealth, HasArmor
{
    static const int32_t MaxLives = 3;
    int32_t NumLives = MaxLives;
};
```

This means `HasHealth` and `HasArmor` are sub-objects of `Player` and `Player` “is a” `HasHealth` and “is a” `HasArmor`. We use it the same way that we use single-inheritance:

```
// Members of both base structs are accessible
Player p{};
DebugLog(p.Health, p.Armor, p.NumLives); // 100, 20, 3

// Can get a reference to a base struct
HasHealth& hh = p;
DebugLog(hh.Health); // 100

// Can get a pointer to a base struct
HasArmor* ha = &p;
DebugLog(ha->Armor); // 20
```

This explains why there is no `base` keyword in C++: there may be multiple bases. When explicitly referencing a member of a sub-object, we always use its name:

```
Player p{};

// Access members in the HasHealth sub-object
DebugLog(p.HasHealth::Health); // 100

// Access members in the HasArmor sub-object
DebugLog(p.HasArmor::Armor); // 20

// Access members in the Player sub-object
DebugLog(p.Player::NumLives); // 3
```

Now let's re-introduce the `LogLifecycle` utility struct and see how multiple inheritance impacts the order of constructors and destructors:

```
struct LogLifecycle
{
    const char* str;

    LogLifecycle(const char* str)
        : str(str)
    {
        DebugLog(str, "constructor");
    }

    ~LogLifecycle()
    {
        DebugLog(str, "destructor");
    }
};

struct HasHealth
{
    LogLifecycle a{"HasHealth::a"};
    LogLifecycle b{"HasHealth::b"};

    static const int32_t MaxHealth = 100;
    int32_t Health = MaxHealth;

    HasHealth()
```

```
    {
        DebugLog("HasHealth constructor");
    }

    ~HasHealth()
    {
        DebugLog("HasHealth destructor");
    }
};

struct HasArmor
{
    LogLifecycle a{"HasArmor::a"};
    LogLifecycle b{"HasArmor::b"};

    static const int32_t MaxArmor = 20;
    int32_t Armor = MaxArmor;

    HasArmor()
    {
        DebugLog("HasArmor constructor");
    }

    ~HasArmor()
    {
        DebugLog("HasArmor destructor");
    }
};
```

```
struct Player : HasHealth, HasArmor
{
    LogLifecycle a{"Player::a"};
    LogLifecycle b{"Player::b"};

    static const int32_t MaxLives = 3;
    int32_t NumLives = MaxLives;

    Player()
    {
        DebugLog("Player constructor");
    }

    ~Player()
    {
        DebugLog("Player destructor");
    }
};

void Foo()
{
    Player p{};
    DebugLog("--after variable declaration--");
} // Note: destructor of 'p' called here

// Logs printed:
//   HasHealth::a, constructor
```

```
//   HasHealth::b, constructor
//   HasHealth constructor
//   HasArmor::a, constructor
//   HasArmor::b, constructor
//   HasArmor constructor
//   Player::a, constructor
//   Player::b, constructor
//   Player constructor
//   --after variable declaration--
//   Player destructor
//   Player::b, destructor
//   Player::a, destructor
//   HasArmor destructor
//   HasArmor::b, destructor
//   HasArmor::a, destructor
//   HasHealth destructor
//   HasHealth::b, destructor
//   HasHealth::a, destructor
```

We see here that base structs' constructors are called in the order that they're derived from: `HasHealth` then `HasArmor` in this example. Their destructors are called in the reverse order. This is analogous to data members, which are constructed in declaration order and destructed in the reverse order.

Multiple inheritance can introduce an ambiguity known as the "diamond problem," referring to the shape of the inheritance hierarchy. For example, consider these structs:

```
struct Top
{
    const char* Id;

    Top(const char* id)
        : Id(id)
    {
    }
};

struct Left : Top
{
    const char* Id = "Left";

    Left()
        : Top("Top of Left")
    {
    }
};

struct Right : Top
{
    const char* Id = "Right";

    Right()
        : Top("Top of Right")
    {
    }
```

```
};

struct Bottom : Left, Right
{
    const char* Id = "Bottom";
};
```

Given a `Bottom`, it's easy to refer to its `Id` and the `Id` of the `Left` and `Right` sub-objects:

```
Bottom b{};
DebugLog(b.Id); // Bottom
DebugLog(b.Left::Id); // Left
DebugLog(b.Right::Id); // Right
```

However, both the `Left` and `Right` sub-objects have a `Top` sub-object. This is therefore ambiguous and causes a compiler error:

```
Bottom b{};

// Compiler error: ambiguous. Top sub-object of Left or
Right?
DebugLog(b.Top::Id);
```

To disambiguate, we need to explicitly refer to either the `Left` or `Right` sub-object. One way is to take a reference to these sub-objects:

```
Bottom b{};
Left& left = b;
DebugLog(left.Top::Id); // Top of Left
Right& right = b;
DebugLog(right.Top::Id); // Top of Right
```

Virtual Inheritance

If we don't want our `Bottom` object to include two `Top` sub-objects then we can use "virtual inheritance" instead. This is just like normal inheritance, except that the compiler will generate only one sub-object for a common base struct. We enable it by adding the keyword `virtual` before the name of the struct we're deriving from:

```
struct Top
{
    const char* Id = "Top Default";
};

struct Left : virtual Top
{
    const char* Id = "Left";
};

struct Right : virtual Top
{
    const char* Id = "Right";
};

struct Bottom : virtual Left, virtual Right
{
    const char* Id = "Bottom";
};

// Top refers to the same sub-object in Bottom, Left, and
```

```
Right
Bottom b{};
Left& left = b;
Right& right = b;
DebugLog(b.Top::Id); // Top Default
DebugLog(left.Top::Id); // Top Default
DebugLog(right.Top::Id); // Top Default

// Changing Left's Top changes the one and only Top sub-
object
left.Top::Id = "New Top of Left";
DebugLog(b.Top::Id); // New Top of Left
DebugLog(left.Top::Id); // New Top of Left
DebugLog(right.Top::Id); // New Top of Left

// Same with Right's Top
right.Top::Id = "New Top of Right";
DebugLog(b.Top::Id); // New Top of Right
DebugLog(left.Top::Id); // New Top of Right
DebugLog(right.Top::Id); // New Top of Right
```

Note that virtual inheritance and regular inheritance can be mixed. In this we get one sub-object for all the common `virtual` base structs and one sub-object *each* for all the non-`virtual` base structs. For example, this `Bottom` has two `Top` sub-objects: one for the `virtual` inhertiance via `Left` and `Right` and one for the non-`virtual` inheritance via `Middle`:

```
struct Top
{
    const char* Id = "Top Default";
};

struct Left : virtual Top
{
    const char* Id = "Left";
};

struct Middle : Top
{
    const char* Id = "Middle";
};

struct Right : virtual Top
{
    const char* Id = "Right";
};

struct Bottom : virtual Left, Middle, virtual Right
{
    const char* Id = "Bottom";
};

// Top refers to the same sub-object in Bottom, Left, and
Right
// It does not refer to Middle's Top sub-object
```

```
Bottom b{};
Left& left = b;
Middle& middle = b;
Right& right = b;
DebugLog(left.Top::Id); // Top Default
DebugLog(middle.Top::Id); // Top Default
DebugLog(right.Top::Id); // Top Default

// Changing Left's Top changes the virtual Top sub-object
left.Top::Id = "New Top of Left";
DebugLog(left.Top::Id); // New Top of Left
DebugLog(middle.Top::Id); // Top Default (note: not
changed)
DebugLog(right.Top::Id); // New Top of Left

// Same with Right's Top
right.Top::Id = "New Top of Right";
DebugLog(left.Top::Id); // New Top of Right
DebugLog(middle.Top::Id); // Top Default (note: not
changed)
DebugLog(right.Top::Id); // New Top of Right

// Changing Middle's Top changes the non-virtual Top sub-
object
middle.Top::Id = "New Top of Middle";
DebugLog(left.Top::Id); // New Top of Right (note: not
changed)
DebugLog(middle.Top::Id); // New Top of Middle
```

```
DebugLog(right.Top::Id); // New Top of Right (note: not changed)
```

Virtual Functions

Member functions in C++ may be virtual. This is directly analogous to virtual methods in C#. Here's how a base `Weapon` struct's `Attack` member function can be overridden by a derived `Bow` struct's `Attack` member function:

```
struct Enemy
{
    int32_t Health = 100;
};

struct Weapon
{
    int32_t Damage = 0;

    explicit Weapon(int32_t damage)
    {
        Damage = damage;
    }

    virtual void Attack(Enemy& enemy)
    {
        enemy.Health -= Damage;
    }
};

struct Bow : Weapon
```

```
{
    Bow(int32_t damage)
        : Weapon(damage)
    {
    }

    virtual void Attack(Enemy& enemy)
    {
        enemy.Health -= Damage;
    }
};

Enemy enemy{};
DebugLog(enemy.Health); // 100

Weapon weapon{10};
weapon.Attack(enemy);
DebugLog(enemy.Health); // 90

Bow bow{20};
bow.Attack(enemy);
DebugLog(enemy.Health); // 70

Weapon& weaponRef = bow;
weaponRef.Attack(enemy);
DebugLog(enemy.Health); // 50

Weapon* weaponPointer = &bow;
```

```
weaponPointer->Attack(enemy);
DebugLog(enemy.Health); // 30
```

Notice that the default behavior in C++ is for a member function is to override a base struct's `virtual` function. This differs from C# where the default is to create a new function with the same name, similar to if the `new` keyword were used to declare the method.

To override in C#, we must use the `override` keyword. In C++, this is optional and mainly used so they compiler will generate an error as a reminder in case our overriding member function no longer matches with a virtual function in a base struct. The `override` keyword comes *after* the function signature, not before as in C#:

```
struct Bow : Weapon
{
    Bow(int32_t damage)
        : Weapon(damage)
    {
    }

    virtual void Attack(Enemy& enemy) override
    {
        enemy.Health -= Damage;
    }

    // Compiler error: no base struct has this virtual
function to override
    virtual float SimulateAttack(Enemy& enemy) override
    {
```

```
        return enemy.Health - Damage;
    }
};
```

C++ also has a replacement for C#'s `abstract` keyword for when we don't want to provide an implementation of a member function at all. These member functions are called "pure virtual" and have an `= 0` after them instead of a body:

```
struct Weapon
{
    int32_t Damage = 0;

    explicit Weapon(int32_t damage)
    {
        Damage = damage;
    }

    virtual void Attack(Enemy& enemy) = 0;
};
```

Like in C# where the presence of any `abstract` methods means we must tag the class itself `abstract`, a C++ class with any pure virtual member functions is implicitly abstract and therefore can't be instantiated:

```
// Compiler error: Weapon is an abstract struct and can't
be instantiated
```

```
Weapon weapon{10};
```

Because, unlike C#, overloaded operators are non-`static`, they too may be `virtual`. In this example, the `+=` operator levels up a `Weapon`:

```
struct Weapon
{
    int32_t Damage = 0;

    explicit Weapon(int32_t damage)
    {
        Damage = damage;
    }

    virtual void operator+=(int32_t numLevels)
    {
        Damage += numLevels;
    }
};

struct Bow : Weapon
{
    int32_t Range;

    Bow(int32_t damage, int32_t range)
        : Weapon(damage), Range(range)
    {
    }
```

```
    virtual void operator+=(int32_t numLevels) override
    {
        // Explicitly call base struct's overloaded
operator
        Weapon::operator+=(numLevels);

        Range += numLevels;
    }
};

Bow bow{20, 10};
DebugLog(bow.Damage, bow.Range); // 20, 10

bow += 5;
DebugLog(bow.Damage, bow.Range); // 25, 15

Weapon& weaponRef = bow;
weaponRef += 5;
DebugLog(bow.Damage, bow.Range); // 30, 20
```

Also unlike C#, user-defined conversion operators are non-`static` so they too may be `virtual`:

```
struct Weapon
{
    int32_t Damage = 0;
```

```
    explicit Weapon(int32_t damage)
    {
        Damage = damage;
    }

    virtual operator int32_t()
    {
        return Damage;
    }
};

struct Bow : Weapon
{
    int32_t Range;

    Bow(int32_t damage, int32_t range)
        : Weapon(damage), Range(range)
    {
    }

    virtual operator int32_t() override
    {
        // Explicitly call base struct's user-defined
conversion operator
        return Weapon::operator int32_t() + Range;
    }
};
```

```
Bow bow{20, 10};
Weapon& weaponRef = bow;
int32_t bowVal = bow;
int32_t weaponRefVal = weaponRef;
DebugLog(bowVal, weaponRefVal); // 30, 30
```

Finally, destructors may be `virtual`. This is very common as it ensures that derived struct destructors will always be called:

```
struct ReadOnlyFile
{
    FILE* ReadFileHandle;

    ReadOnlyFile(const char* path)
    {
        ReadFileHandle = fopen(path, "r");
    }

    virtual ~ReadOnlyFile()
    {
        fclose(ReadFileHandle);
        ReadFileHandle = nullptr
    }
};

struct FileCopier : ReadOnlyFile
{
    FILE* WriteFileHandle;
```

```
    FileCopier(const char* path, const char*
writeFilePath)
        : ReadOnlyFile(path)
    {
        WriteFileHandle = fopen(writeFilePath, "w");
    }

    virtual ~FileCopier()
    {
        fclose(WriteFileHandle);
        WriteFileHandle = nullptr;
    }
};

FileCopier copier("/path/to/input/file",
"/path/to/output/file");

// Calling a virtual destructor on a base struct
// Calls the derived struct's destructor
// If this was non-virtual, only the ReadOnlyFile
destructor would be called
ReadOnlyFile& rof = copier;
rof.~ReadOnlyFile();
```

A pure virtual destructor is also a common way of marking a struct as abstract when no other members are good candidates to be made pure virtual. This is like marking a class `abstract` in C# without marking any of its contents `abstract`.

Stopping Inheritance and Overrides

C# has the `sealed` keyword that can be applied to a class to stop other classes from deriving it. C++ has the `final` keyword for this purpose. It's placed after the struct name and before any base struct names:

```
// OK to derive from Vector1
struct Vector1
{
    float X;
};

// Compiler error if deriving from Vector2
struct Vector2 final : Vector1
{
    float Y;
};

// Compiler error: Vector2 is final
struct Vector3 : Vector2
{
    float Z;
};
```

The `sealed` keyword in C# can also be applied to methods and properties that `override` to stop further overriding. The `final` keyword in C++ also serves this purpose. Like the `override` keyword, it's placed after the member function signature. The two

keywords can be used together as they have different, but related meanings.

```
struct Vector1
{
    float X;

    // Allows overriding
    virtual void DrawPixel(float r, float g, float b)
    {
        GraphicsLibrary::DrawPixel(X, 0, r, g, b);
    }
};

struct Vector2 : Vector1
{
    float Y;

    // Overrides DrawPixel in Vector1
    // Stops overriding in derived structs
    virtual void DrawPixel(float r, float g, float b)
override final
    {
        GraphicsLibrary::DrawPixel(X, Y, r, g, b);
    }
};

struct Vector3 : Vector2
```

```
{
    float Z;

    // Compiler error: DrawPixel in base struct (Vector2)
is final
    virtual void DrawPixel(float r, float g, float b)
override
    {
        GraphicsLibrary::DrawPixel(X/Z, Y/Z, r, g, b);
    }
};
```

C# Equivalency

We've already seen the C++ equivalents for C# concepts like `abstract` and `sealed` classes as well as `abstract`, `virtual`, `override`, and `sealed` methods. C# has several other features that have no explicit C++ equivalent. Instead, these are idiomatically implemented with general struct features.

First, C# has a dedicated `interface` concept. It's like a base class that can't have fields, can't have non-abstract methods, is abstract, and can be multiply inherited by classes. In C++, we always have multiple inheritance so all we need to do is not add any data members or non-abstract member functions:

```
// Like an interface:
// * Has no data members
// * Has no non-abstract member functions
// * Is abstract (due to Log being pure virtual)
// * Enables multiple inheritance (always enabled in C++)
struct ILoggable
{
    virtual void Log() = 0;
};

// To "implement" an "interface," just derive and
override all member functions
struct Vector2 : ILoggable
{
    float X;
    float Y;
```

```
    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }

    virtual void Log() override
    {
        DebugLog(X, Y);
    }
};

// Use an "interface," not a "concrete class"
void LogTwice(ILoggable& loggable)
{
    loggable.Log();
    loggable.Log();
}

Vector2 vec{2, 4};
LogTwice(vec); // 2, 4 then 2, 4
```

Next we have `partial` classes in C#. These allow us to split the contents of a class across multiple files. We can mimic this in C++ in at least two ways. First, and far more commonly, we put the struct's definition in a header file (e.g. `.h`) and split its member definitions across multiple translation unit (e.g. `.cpp`) files.

```
// In Player.h
struct Player
{
    const static int32_t MaxHealth = 100;
    const static int32_t MaxLives = 3;
    int32_t Health = MaxHealth;
    int32_t NumLives = MaxLives;
    float PosX = 0;
    float PosY = 0;
    float DirX = 0;
    float DirY = 0;
    float Speed = 0;

    void TakeDamage(int32_t amount);
    void Move(float time);
};

// In PlayerCombat.cpp
#include "Player.h"
void Player::TakeDamage(int32_t amount)
{
    Health -= amount;
}

// In PlayerMovement.cpp
#include "Player.h"
void Player::Move(float time)
{
```

```
    float distance = Speed * time;
    PosX += DirX * distance;
    PosY += DirY * distance;
}

// In Game.cpp
#include "Player.h"
Player player;
player.DirX = 1;
player.Speed = 1;
player.TakeDamage(10);
DebugLog(player.Health); // 90
player.Move(5);
DebugLog(player.PosX, player.PosY); // 5, 0
```

Another, much less common, approach is to use virtual inheritance to compose multiple structs into one:

```
// In PlayerShared.h
struct PlayerShared
{
    const static int32_t MaxHealth = 100;
    const static int32_t MaxLives = 3;
    int32_t Health = MaxHealth;
    int32_t NumLives = MaxLives;
    float PosX = 0;
    float PosY = 0;
    float DirX = 0;
```

```
    float DirY = 0;
    float Speed = 0;
};

// In PlayerCombat.h
#include "PlayerShared.h"
struct PlayerCombat : virtual PlayerShared
{
    void TakeDamage(int32_t amount)
    {
        Health -= amount;
    }
};

// In PlayerMovement.h
#include "PlayerShared.h"
struct PlayerMovement : virtual PlayerShared
{
    void Move(float time)
    {
        float distance = Speed * time;
        PosX += DirX * distance;
        PosY += DirY * distance;
    }
};

// In Player.h
#include "PlayerCombat.h"
```

```
#include "PlayerMovement.h"
struct Player : virtual PlayerCombat, virtual
PlayerMovement
{
};

// In Game.cpp
#include "Player.h"
Player player;
player.DirX = 1;
player.Speed = 1;
player.TakeDamage(10);
DebugLog(player.Health); // 90
player.Move(5);
DebugLog(player.PosX, player.PosY); // 5, 0
```

This approach allows the parts (`PlayerCombat` and `PlayerMovement`) to be re-composed to form other types, such as a `StationaryPlayer` that can fight but not move:

```
// In StationaryPlayer.h
#include "PlayerCombat.h"
struct StationaryPlayer : virtual PlayerCombat
{
};

// In Game.cpp
#include "StationaryPlayer.h"
```

```
StationaryPlayer stationary;
stationary.TakeDamage(10); // OK, Health now 90
stationary.Move(5); // Compiler error: Move isn't a
member function
```

Finally, all C# classes have `System.Object` as their ultimate base class. C# structs, and other value types, are implicitly “boxed” to `System.Object` when used in particular ways such as calling the `GetHashCode` method. C++ has no ultimate base struct like this, but one can be created and all other structs can derive from it.

```
// In Object.h
// Ultimate base struct
struct Object
{
    virtual int32_t GetHashCode()
    {
        return HashBytes((char*)this, sizeof(*this));
    }
};

// In Player.h
#include "Object.h"
// Derives from ultimate base struct: Object
struct Player : Object
{
    const static int32_t MaxHealth = 100;
    const static int32_t MaxLives = 3;
    int32_t Health = MaxHealth;
```

```
    int32_t NumLives = MaxLives;
    float PosX = 0;
    float PosY = 0;
    float DirX = 0;
    float DirY = 0;
    float Speed = 0;

    void TakeDamage(int32_t amount);
    void Move(float time);

    // Can override if desired, like in C#
    virtual int32_t GetHashCode() override
    {
        return 123;
    }
};

// In Vector2.h
#include "Object.h"
// Derives from ultimate base struct: Object
struct Vector2 : Object
{
    float X;
    float Y;

    // Can NOT override if desired, like in C#
    // virtual int32_t GetHashCode() override
};
```

```
// Can pass any struct to this
// Because we made every struct derive from Object
void LogHashCode(Object& obj)
{
    DebugLog(obj.GetHashCode());
}

// Can pass a Player because it derives from Object
Player player;
LogHashCode(player);

// Can pass a Vector2 because it derives from Object
Vector2 vector;
LogHashCode(vector);
```

Generic solutions for boxing are likewise possible and we'll cover those techniques later in the book.

Conclusion

C++ struct inheritance is in many ways a superset of C# class inheritance. It goes above and beyond with support for multiple inheritance, virtual inheritance, virtual overloaded operators, virtual user-defined conversion functions, and skip-level sub-object specifications like `Level1::X` from within `Level3`.

In other ways, C++ inheritance is more stripped down than C# inheritance. It doesn't have dedicated support for interfaces or `partial` classes and it doesn't mandate an ultimate base struct like `Object` in C#. To recover these features, we rely on the extended feature set's increased flexiblity to essentially build our own interfaces, `partial` classes, and `Object` type.

15. Struct and Class Permissions

Access Specifiers

Like in C#, the members of structs in C++ can have their access level changed by the `public`, `protected`, and `private` access specifiers. It's written a little differently in C++ though. Instead of being included like a modifier of a single member (e.g. `public void Foo() {}`), access specifiers in C++ are written like a label (e.g. `public:`) and apply until the next access specifier:

```
struct Player
{
// The default access specifier is public, so TakeDamage
is public
    void TakeDamage(int32_t amount)
    {
        Health -= amount;
    }

// Change the access specifier to private
private:

// Health is private
    int32_t Health;

// NumLives is private
    int32_t NumLives;
```

```
// Change the access specifier back to public
public:

// Heal is public
    void Heal(int32_t amount)
    {
        Health += amount;
    }

// GetExtraLife is public
    void GetExtraLife()
    {
        NumLives++;
    }
};
```

While uncommon in C++, we can make this feel more like C# by explicitly adding the access specifier before every member:

```
struct Player
{
    public: void TakeDamage(int32_t amount)
    {
        Health -= amount;
    }

    private: int32_t Health;
    private: int32_t NumLives;
```

```
    public: void Heal(int32_t amount)
    {
        Health += amount;
    }

    public: void GetExtraLife()
    {
        NumLives++;
    }
};
```

The meaning of `public`, `private`, and `protected` are similar to C#:

Access Specifier	Member Accessibile From
`public`	Anywhere
`protected`	Only within the struct and in derived structs
`private`	Only within the struct

Unlike C#, access specifiers may also be applied when deriving from structs:

```
struct PublicPlayer : public Player
{
};

struct ProtectedPlayer : protected Player
```

```
{
};

struct PrivatePlayer : private Player
{
};

// Default inheritance access specifier is public
struct DefaultPlayer : Player
{
};
```

The inheritance access specifier maps the member access levels in the base struct to access levels in the derived struct:

	Inherit `public`	**Inherit `protected`**	**Inherit `private`**
Base `public`	`public`	`protected`	`private`
Base `private`	`private`	`private`	`private`
Base `protected`	`protected`	`protected`	`private`

This means that `ProtectedPlayer` and `PrivatePlayer` have hidden the `public` members of `Player` from outside code:

```
PublicPlayer pub{};
pub.Heal(10); // OK: Heal is public
```

```
ProtectedPlayer prot{};
prot.Heal(10); // Compiler error: Heal is protected

PrivatePlayer priv{};
priv.Heal(10); // Compiler error: Heal is private

DefaultPlayer def{};
def.Heal(10); // OK: Heal is public
```

When a virtual member function is overridden, it may have a different access level than the member function it overrides. In this case, the access level is always determined at compile time using the type the member function is being called on. This may be different than the runtime type of the object. That means access specifiers don't support runtime polymorphism.

```
struct Base
{
    virtual void Foo()
    {
        DebugLog("Base Foo");
    }

private:

    virtual void Goo()
    {
        DebugLog("Base Goo");
    }
```

```
};

struct Derived : Base
{
private:

    virtual void Foo() override
    {
        DebugLog("Derived Foo");
    }

public:

    virtual void Goo() override
    {
        DebugLog("Derived Goo");
    }
};

// These calls use the access specifiers in Base
Base b;
b.Foo(); // "Base Foo"
//b.Goo(); // Compiler error: Goo is private

// These calls use the access specifiers in Derived
Derived d;
//d.Foo(); // Compiler error: Foo is private
d.Goo(); // "Derived Goo"
```

```
// These calls use the access specifiers in Base, even
though the runtime object
// is a Derived
Base& dRef = d;
dRef.Foo(); // "Derived Foo"
//dRef.Goo(); // Compiler error: Goo is private in Base
```

When using virtual inheritance, the most accessible path through the derived classes is used to determine access level:

```
struct Top
{
    int32_t X = 123;
};

// X is private due to private inheritance
struct Left : private virtual Top
{
};

// X is public due to public inheritance
struct Right : public virtual Top
{
};

// X is public due to public inheritance via Right
// This takes precedence over private inheritance via
```

```
Left
struct Bottom : virtual Left, virtual Right
{
};

Top top{};
DebugLog(top.X); // 123

Left left{};
//DebugLog(left.X); // Compiler error: X is private

Right right{};
DebugLog(right.X); // 123

// Accessing X goes through Right
Bottom bottom{};
DebugLog(bottom.X); // 123
```

It's important to note that access levels may change the layout of the struct's non-static data members in memory. While the data members are guaranteed to be laid out sequentially, perhaps with padding between them, this is only true of data members *of the same access level*. For example, the compiler may choose to lay out all the `public` data members then all the `private` data members or to mix all the data members regardless of their access level:

```
struct Mixed
{
    private: int32_t A = 1;
```

```
    public: int32_t B = 2;
    private: int32_t C = 3;
    public: int32_t D = 4;
};

// Some possible layouts of Mixed:
//   Ignore access level: A, B, C, D
//   Private then public: A, C, B, D
//   Public then private: B, D, A, C
```

Structs, including all of the examples above, use `public` as their default access level. That applies to their members and their inheritance. To make `private` the default, replace the keyword `struct` with `class`:

```
class Player
{
    int32_t Health = 0;
};

Player player{};
DebugLog(player.Health); // Compiler error: Health is
private
```

That's right: *classes are just structs with a different default access level!*

They're so compatible that we can even declare them as `struct` and define them as `class` or visa versa:

```
struct Player;
class Player
{
    int32_t Health = 0;
};

class Weapon;
struct Weapon
{
    int32_t Damage = 0;
};
```

The choice of which to use is mostly up to convention. The `struct` keyword is typically used when all or the majority of members will be `public`. The `class` keyword is typically used when all or the majority of members will be `private`.

As far as terminology, it's typical to say just "classes" or "structs" rather than "classes and structs" since the two concepts are essentially the same. For example, "structs can have constructors" implies that classes can also have constructors. All of the previous articles about structs in this book apply equally to classes.

Friendship

C++ provides a way for structs to explicitly grant complete access to their members regardless of what the access level would otherwise be. To do so, the struct adds a `friend` declaration to its definition stating the name of the function or struct that it wants to grant access to:

```
class Player
{
    // Private due to the default for classes
    int64_t Id = 123;
    int32_t Points = 0;

    // Make the PrintId function a friend
    friend void PrintId(const Player& player);

    // Make the Stats class a friend
    friend class Stats;
};

void PrintId(const Player& player)
{
    // Can access Id and Points members because PrintId
is a friend of Player
    DebugLog(player.Id, "has", player.Points, "points");
}

// It's OK that Stats is actually a struct, not a class
```

```
struct Stats
{
    static int32_t GetTotalPoints(Player* players,
int32_t numPlayers)
    {
        int32_t totalPoints = 0;
        for (int32_t i = 0; i < numPlayers; ++i)
        {
            // Can access Points because Stats is a
friend of Player
            totalPoints += players[i].Points;
        }
        return totalPoints;
    }
};

Player p;
PrintId(p); // 123 has 0 points

int32_t totalPoints = Stats::GetTotalPoints(&p, 1);
DebugLog(totalPoints); // 0
```

It's so common for structs to be friends with `inline` functions in particular that C++ provides a shortcut to defining the `inline` function and declaring it as a friend:

```
class Player
{
```

```
    int64_t Id = 123;
    int32_t Points = 0;

    // Make the PrintId inline function and make it a
friend
    friend void PrintId(const Player& player)
    {
        // Can access Id and Points members
        // because PrintId is a friend of Player
        DebugLog(player.Id, "has", player.Points,
"points");
    }
};
```

Even though the definition of `PrintId` appears within the definition of `Player`, like a member function would, it is still a normal function and *not* a member function. We can see this when calling it:

```
// Call like a normal function
Player p;
PrintId(p); // 123 has 0 points

// Call like a member function
p.PrintId(); // Compiler error: Player doesn't have a
PrintId member
```

Also, note that friendship is not inherited and a friend-of-a-friend is not a friend:

```
class Player
{
    int32_t Points = 0;
    friend class Stats;
};

struct Stats
{
    friend class PointStats;
};

struct PointStats : Stats
{
    static int32_t GetTotalPoints(Player* players,
int32_t numPlayers)
    {
        int32_t totalPoints = 0;
        for (int32_t i = 0; i < numPlayers; ++i)
        {
            // Compiler error: can't access Points
because PointStats is NOT
            // a friend of Player even though it inherits
from Stats and is
            // a friend of Stats. Only Stats is a friend.
            totalPoints += players[i].Points;
        }
        return totalPoints;
```

```
    }
};
```

Const and Mutable

As we've seen throughout the book and touched on lightly, types in C++ may be qualified with the `const` keyword. We can put this qualifier on any use of a type: local variables, global variables, data members, function parameters, return values, pointers, and references.

At its most basic, `const` means we can't re-assign:

```
// x is a const int
const int32_t x = 123;

// Compiler error: can't assign to a const variable
x = 456;
```

The `const` keyword can be placed on the left or right of the type. This is known as "west `const`" and "east `const`" and both are in common usage. The placement makes no difference in this case as both result in a `const` type.

```
const int32_t x = 123; // "West const" version of a
constant int32_t
int32_t const y = 456; // "East const" version of a
constant int32_t
DebugLog(x, y); // 123, 456
```

For even slightly more complicated types, the ordering matters more. Consider a pointer type:

```
const char* str = "hello";
```

There are two potential meanings of `const char* str`:

1. A non-`const` pointer to a `const` `char`
2. A `const` pointer to a non-`const` `char`

That is to say, one of these two will be a compiler error:

```
// Compiler error if (1) because this would change the
char
*str = 'H';

// Compiler error if (2) because this would change the
pointer
str = "goodbye";
```

The rule is that `const` modifies what's immediately to its left. If there's nothing to its left, it modifies what's immediately to its right.

Because there's nothing left of `const` in `const char* str`, the `const` modifies the `char` immediately to its right. That means the `char` is `const` and the pointer is non-`const`:

```
*str = 'H'; // Compiler error: character is const
str = "goodbye"; // OK: pointer now points to "goodbye"
```

Using "east `const`" eliminates the "nothing to its left" case so it's easier to determine what is `const`:

```
// 'char' is left of 'const', so 'char' is const
char const * str = "hello";
str = "goodbye"; // OK: pointer is non-const
*str = 'H'; // Compiler error: char is const

// '*' is left of 'const', so pointer is const
char * const str = "hello";
str = "goodbye"; // Compiler error: pointer is const
*str = 'H'; // OK: char is non-const

// Both 'char' and '*' are left of a 'const', so both are
const
char const * const str = "hello";
str = "goodbye"; // Compiler error: pointer is const
*str = 'H';// Compiler error: char is const
```

Besides assignment, the `const` keyword also means we can't modify the data members of a `const` struct:

```
struct Player
{
    const int64_t Id;

    Player(int64_t id)
        : Id(id)
```

```
    {
    }
};

Player player{123};

// Compiler error: can't modify data members of a const
struct
//player.Id = 1000;
```

A reference to a `const T` has the type `const T&` and a pointer to a `const T` has the type `const T*`. These can't be assigned to non-`const` references with type `T&` and non-`const` pointers with type `T*` since that would remove the "`const`-ness" of it:

```
const int32_t x = 123;

// Compiler error: const int32_t& incompatible with
int32_t&
int32_t& xRef = x;

// Compiler error: const int32_t* incompatible with
int32_t*
int32_t* xPtr = &x;
```

Member functions may also be `const` by putting the keyword after the function signature, similar to where we'd put `override`:

```
class Player
{
    int32_t Health = 100;

public:

    // GetHealth is a const member function
    int32_t GetHealth() const
    {
        return Health;
    }
};
```

A `const` member function of a struct `T` is implicitly passed a `const T* this` pointer instead of a (non-`const`) `T* this` pointer. This means all the same restrictions apply and it is not allowed to modify the data members of `this`. This is similar to `readonly` instance members in C# 8.0.

We're also prohibited from calling non-`const` member functions on a `const` struct, either from inside or outside the struct:

```
class Player
{
    int32_t Health = 100;

public:

    int32_t GetHealth() const
```

```
    {
        // Compiler error: can't call non-const member
function from const
        //                member function because 'this'
is a 'const Player*'
        TakeDamage(1);

        return Health;
    }

    void TakeDamage(int32_t amount)
    {
        Health -= amount;
    }
};

Player player{};
const Player& playerRef = player;

// Compiler error: can't call non-const TakeDamage on
const reference
playerRef.TakeDamage();

// OK: GetHealth is const
DebugLog(playerRef.GetHealth()); // 100
```

To opt-out of this restriction and allow a particular data member to be treated as non-`const` by a `const` member function, we use the `mutable` keyword:

```
class File
{
    FILE* Handle;

    // Can be modified by const member functions
    mutable long Size;

public:
    File(const char* path)
    {
        Handle = fopen(path, "r");
        Size = -1;
    }

    ~File()
    {
        fclose(Handle);
    }

    // A const member function
    long GetSize() const
    {
        if (Size < 0)
        {
            long oldPos = ftell(Handle);
            fseek(Handle, 0, SEEK_END);
            Size = ftell(Handle); // OK: Size is mutable
            fseek(Handle, oldPos, SEEK_SET);
```

```
        }
        return Size;
    }
};
```

The `mutable` keyword is commonly used with cached values like the above caching of the file's size. For example, it means that the relatively-expensive file I/O operations only need to be performed one time regardless of how many times `GetSize` is called:

```
const File f{"/path/to/file"};
for (int32_t i = 0; i < 1000; ++i)
{
    // Can call GetSize on const File because GetSize is
const
    // GetSize can update Size because Size is mutable
    // Only the first call does any file I/O: saves 999
file I/O operations
    DebugLog(f.GetSize());
}
```

The following table contrasts C++'s `const` keyword with C#'s `const` and `readonly` keywords:

Factor	**C++ `const`**	**C# `const`**	**C# `readonly`**
Types	Any	Numbers, strings, booleans	Any

Factor	**C++ `const`**	**C# `const`**	**C# `readonly`**
Applicability	Everywhere	Fields, local variables	Fields, references
Can assign to field	Only `mutable`	N/A	No
Can set field of field	Only `mutable`	N/A	Structs no, Classes yes
Can set field of reference	Only `mutable`	N/A	Yes, but doesn't change referenced structs
Can call function of field	Only `const` function	N/A	Yes, but doesn't change field of structs
Can call function of reference	Only `const` function	N/A	Yes, but doesn't change referenced structs

Generally, C++'s 'const' provides more consistent and thorough immutability than C#'s `readonly`, which is its closest equivalent.

C# Equivalency

C# has three more access levels than C++:

- `internal`
- `protected internal`
- `private protected`

Access Specifier	Member Accessibile From
`internal`	Within same assembly
`protected internal`	Within same assembly or in derived classes
`private protected`	Within same assembly or in derived classes within same assembly

C++ has no concept of "assemblies" like .NET DLLs, so none of these apply. They can, however, be simulated in C++.

One solution employs a combination of `friend` classes and withholding some of the library's header files to simulate `internal`:

```
// Player.h: distributed with library (e.g. in 'public' directory)
class Player
{
    // "Internal" access is granted via a particular struct
    friend struct PlayerInternal;
```

```
    // "Internal" data members are private instead
    int64_t Id;

public:

    // Public library API
    int64_t GetId() const
    {
        return Id;
    }
};

// PlayerInternal.h: NOT distributed with library (e.g.
in 'internal' directory)
#include "Player.h"
struct PlayerInternal
{
    // Functions of the friend struct provide access to
"internal" data members
    static int64_t& GetId(Player& player)
    {
        return player.Id;
    }
};

// Library.cpp: code written inside the library
#include "Player.h"
```

```
#include "PlayerInternal.h"
Player player{};
PlayerInternal::GetId(player) = 123; // OK
DebugLog(player.GetId()); // OK

// User.cpp: code written by the user of the library
#include "Player.h"
Player player{};
PlayerInternal::GetId(player) = 123; // Compiler error:
undefined PlayerInternal
DebugLog(player.GetId()); // OK
```

Variants of this approach can be created to simulate `protected internal` and `private protected`.

Conclusion

Once again, the two languages have quite a lot of overlap but also many differences. They both feature `public`, `protected`, and `private` access specifiers with roughly the same meaning. C# also has `internal`, `protected internal`, and `private protected`, which C++ needs to simulate. C++ has inheritance access specifiers, friend functions, and friend classes while C# doesn't.

C++ `const` serves a similar role to C# `readonly` and `const`, but applies much more broadly than either. Allowing it on function parameters and return values as well as arbitrary variables, pointers, and references is a feature that has major impact on how C++ code is designed. A lot of C++ codebases make as many variables, references, and functions as possible `const` to enforce immutability with compiler errors. The `mutable` keyword offers flexibility, such as when implementing caches.

The last gigantic difference between the languages is that structs and classes are the same in C++. The only difference is the keyword used and the default access level. In C#, structs and classes are extremely different. For example, C# structs don't support inheritance or default constructors and there are are no `readonly class` types or unmanaged classes. C++ structs and classes provide almost the combined feature set of C# structs, classes, and interfaces as well as being used to build other language features such as delegates.

16. Struct and Class Wrapup

User-Defined Literals

C++ supports creating our own literals, with some limitations. These are used to create instances of structs or other types in a similar manner to user-defined conversion operators. They're just converting from literals rather than existing objects.

Here are the kinds of literals we can create:

Name	Example
Decimal literal	`123_suffix`
Octal literal	`0123_suffix`
Hexadecimal literal	`0x123_suffix`
Binary literal	`0b123_suffix`
Real literal	`0.123_suffix`
Character literal	`'c'_suffix`
String literal	`"c"_suffix`

The suffix can be any valid identifier. To implement the literal, we write an `operator "" _suffix` function that's not part of the struct:

```
Vector2 operator "" _v2(long double val)
{
    Vector2 vec;
    vec.X = val;
```

```
    vec.Y = val;
    return vec;
}
```

Then we call it like this:

```
Vector2 v1 = 2.0_v2;
DebugLog(v1.X, v1.Y); // 2, 2
```

The C++ Standard Library reserves all suffixes that don't start with an `_` for its own use:

```
string greeting = "hello"s;
hours halfHour = 0.5h;
```

Like with other forms of operator overloading, including user-defined conversion operators, it's important to strongly consider how understandable the resulting code will be given its terseness. Regular constructors and member functions may be more easily understood due to explicitly stating the type.

Still, there are situations where the brevity and expressiveness may come in handy. This is especially the case for codebases that make heavy use of `auto`:

```
// User-defined literals require _less_ typing with auto
auto a = Vector2{2.0f};
auto b = 2.0f_v2;
```

```
// User-defined literals require _more_ typing without
auto
Vector2 a{2.0f};
Vector2 b = 2.0f_v2;
```

Local Classes

A local class (or struct) is one that is defined within the body of a function:

```
void Foo()
{
    struct Local
    {
        int32_t Val;

        Local(int32_t val)
            : Val(val)
        {
        }
    };

    Local ten{10};
    DebugLog(ten.Val); // 10
}
```

Local classes are regular classes in most ways, but have a few limitations. First, their member functions have to be defined within the class definition: we can't split the declaration and the definition.

```
void Foo()
{
    struct Local
```

```
    {
        int32_t Val;

        Local(int32_t val);
    };

    // Compiler error
    // Member function definition must be in the class
definition
    Local::Local(int32_t val)
        : Val(val)
    {
    }
}
```

Second, they can't have static data members but they can have static member functions.

```
void Foo()
{
    struct Local
    {
        int32_t Val;

        // Compiler error
        // Local classes can't have static data members
        static int32_t Max = 100;
```

```
        // OK: local classes can have static member
functions
        static int32_t GetMax()
        {
            return 100;
        }
    };

    DebugLog(Local::GetMax()); // 100
}
```

Third, and finally, they can have friends but they can't declare inline friend functions:

```
class Classy
{
};

void Foo()
{
    struct Local
    {
        // Compiler error
        // Local classes can't define inline friend
functions
        friend void InlineFriend()
        {
        }
```

```
        // OK: local classes can have normal friends
        friend class Classy;
    };
}
```

Like local functions in C#, local classes in C++ are typically used to reduce duplication of code inside the function but are placed inside the function because they wouldn't be useful to code outside the function. It's even common to see local classes without a name when only one instance of them is needed. For example, this local class de-duplicates code that's run on players, enemies, and NPCs without requiring polymorphism:

```
// Three unrelated types: no common base class
struct Player
{
    int32_t Health;
};
struct Enemy
{
    int32_t Health;
};
struct Npc
{
    int32_t Health;
};

int32_t HealToFullIfNotDead(
```

```
    Player* players, int32_t numPlayers,
    Enemy* enemies, int32_t numEnemies,
    Npc* npcs, int32_t numNpcs)
{
    // Anonymous local class
    // Avoids needing to pick a good name
    struct
    {
        // More than just a function wrapped in a class
        // Also has its own state to keep track of
healing
        int32_t NumHealed = 0;

        // Overloaded function call operator
        // Avoids needing to pick a good name
        int32_t operator()(int32_t health)
        {
            // Dead or already at full. No heal.
            if (health <= 0 || health >= 100)
            {
                return health;
            }

            // Damaged. Heal.
            NumHealed++;
            return 100;
        }
    } healer;
```

```
    // The body of each loop reuses the heal code
    for (int32_t i = 0; i < numPlayers; ++i)
    {
        // Call the overloaded function call operator
        players[i].Health = healer(players[i].Health);
    }
    for (int32_t i = 0; i < numEnemies; ++i)
    {
        enemies[i].Health = healer(enemies[i].Health);
    }
    for (int32_t i = 0; i < numNpcs; ++i)
    {
        npcs[i].Health = healer(npcs[i].Health);
    }

    return healer.NumHealed;
}

// One dead, two damaged, one full health for each
const int32_t num = 4;
Player players[num]{{0}, {50}, {75}, {100}};
Enemy enemies[num]{{0}, {50}, {75}, {100}};
Npc npcs[num]{{0}, {50}, {75}, {100}};
int32_t numHealed = HealToFullIfNotDead(
    players, num,
    enemies, num,
    npcs, num);
```

```
DebugLog(numHealed); // 6
DebugLog(
    players[0].Health, players[1].Health,
    players[2].Health, players[3].Health); // 0, 100,
100, 100
DebugLog(
    enemies[0].Health, enemies[1].Health,
    enemies[2].Health, enemies[3].Health); // 0, 100,
100, 100
DebugLog(
    npcs[0].Health, npcs[1].Health,
    npcs[2].Health, npcs[3].Health); // 0, 100, 100, 100
```

Copy and Move Assignment Operators

Along with destructors and some constructors, the compiler will also generate copy and move assignment operators for us.

```
struct Vector2
{
    float X;
    float Y;

    // Compiler generates a copy assignment operator like this:
    // Vector2& operator=(const Vector2& other)
    // {
    //     X = other.X;
    //     Y = other.Y;
    //     return *this;
    // }

    // Compiler generates a move assignment operator like this:
    // Vector2& operator=(const Vector2&& other)
    // {
    //     X = other.X;
    //     Y = other.Y;
    //     return *this;
    // }
};
```

```
void Foo()
{
    Vector2 a{2, 4};
    Vector2 b{0, 0};
    b = a; // Call the compiler-generated copy assignment
operator
    DebugLog(b.X, b.Y); // 2, 4
}
```

It'll do this as long as we don't define the assignment operator ourselves, each non-static data member and base class has an assignment operator, and none of the non-static data members are `const` or references.

Like constructors and destructors, we can use `= default` and `= delete` to override the default behavior and either force the compiler to generate one or force it to not generate one.

```
struct Vector2
{
    float X;
    float Y;

    Vector2& operator=(const Vector2& other) = delete;
};

void Foo()
{
```

```
    Vector2 a{2, 4};
    Vector2 b{0, 0};
    b = a; // Compiler error: copy assignment operator is
deleted
    DebugLog(b.X, b.Y); // 2, 4
}
```

Unions

We've seen how the `class` keyword can be used instead of `struct` to change the default access level from `public` to `private`. Similarly, C++ provides the `union` keyword to change the data layout of the struct. Instead of making the struct big enough to fit all of the non-static data members, a union is just big enough to fit the *largest* non-static data member.

```
union FloatBytes
{
    float Val;
    uint8_t Bytes[4];
};

void Foo()
{
    FloatBytes fb;
    fb.Val = 3.14f;
    DebugLog(sizeof(fb)); // 4 (not 8)

    // 195, 245, 72, 64
    DebugLog(fb.Bytes[0], fb.Bytes[1], fb.Bytes[2],
fb.Bytes[3]);

    fb.Bytes[0] = 0;
    fb.Bytes[1] = 0;
    fb.Bytes[2] = 0;
    fb.Bytes[3] = 0;
```

```
    DebugLog(fb.Val); // 0
}
```

Because the non-static data members of a union occupy the same memory space, writing to one writes to the other. In the above example, we can use this to get the bytes that make up a `float` or to manipulate the `float` using integer math on the byte array that it shares memory with.

Note that it is technically undefined behavior to read any non-static data member except the most recently written one. However, nearly all compilers support this as it is a common usage for unions so it is very likely to be safe.

Like local classes, there are some restrictions put on unions. First, unions can't participate in inheritance. That means they can't have any base classes, be a base class themselves, or have any virtual member functions.

```
struct IGetHashCode
{
    virtual int32_t GetHashCode() = 0;
};

// Compiler error: unions can't derive
union Id : IGetHashCode
{
    int32_t Val;
    uint8_t Bytes[4];

    // Compiler error: unions can't have virtual member
```

```
functions
    virtual int32_t GetHashCode() override
    {
        return Val;
    }
};

// Compiler error: can't derive from a union
struct Vec2Bytes : Id
{
};
```

Second, unions can't have non-static data members that are references:

```
union IntRefs
{
    // Compiler error: unions can't have lvalue
references
    int32_t& Lvalue;

    // Compiler error: unions can't have rvalue
references
    int32_t&& Rvalue;
};
```

Third, if any non-static data member of the union has a "non-trivial" copy or move constructor, copy or move assignment operator, or

destructor, then the union's version of that function is deleted by default and needs to be explicitly written.

A struct has a "non-trivial" constructor if it's explicitly written, or if any of the non-static data members have default initializers, or if there are any virtual member functions or base classes, or if any non-static data member or base class has a non-trivial constructor.

A struct has a "non-trivial" destructor if it's explicitly written, virtual, or any non-static data member or base class has a non-trivial destructor.

A struct has a "non-trivial" assignment operator if it's explicitly written, if there are any virtual member functions or base classes, or any non-static data member or base class has a non-trivial assignment operator.

That's a lot of rules, but it's rather uncommon for unions to include types with these kinds of non-trivial functions. Typically they're used for simple primitives, structs, and arrays, like in the above examples. For more advanced usage, we need to keep the rules in mind:

```
// Note: "ctor" is a common abbreviation for
"constructor"
//       Likewise, "dtor" is a common abbreviation for
"destructor"
struct NonTrivialCtor
{
    int32_t Val;

    NonTrivialCtor()
    {
        Val = 100;
```

```
    }

    // Non-trivial copy constructor because it's
explicitly written
    NonTrivialCtor(const NonTrivialCtor& other)
    {
        Val = other.Val;
    }
};

// Union with a non-static data member whose copy
constructor is non-trivial
// The union's copy constructor is deleted by default
union HasNonTrivialCtor
{
    NonTrivialCtor Ntc;
};

// Union with a non-static data member whose copy
constructor is non-trivial
// The union's copy constructor is deleted by default
union HasNonTrivialCtor2
{
    NonTrivialCtor Ntc;

    HasNonTrivialCtor2()
        : Ntc{}
    {
```

```
    }

    // Explicitly write a copy constructor
    HasNonTrivialCtor2(const HasNonTrivialCtor2& other)
        : Ntc{other.Ntc}
    {
    }
};

HasNonTrivialCtor a{};
DebugLog(a.Ntc.Val);

// Compiler error
// Union has a non-static data member with a non-trivial
copy constructor
// Its copy constructor must be written explicitly
HasNonTrivialCtor b{a};
DebugLog(b.Ntc.Val);

HasNonTrivialCtor2 c{};

// OK: copy constructor explicitly written
HasNonTrivialCtor2 d{c};
DebugLog(d.Ntc.Val); // 100
```

Unions can also be "anonymous." Like structs, they can have no name. Unlike structs, they can also have no variable:

```
void Foo()
{
    union
    {
        int32_t Int;
        float Float;
    };
}
```

These are even more restricted than normal unions. They can't have any member functions or static data members and all their data members have to be public. Like unscoped enums, their members are added to whatever scope the union is in: `Foo` in the above example.

```
void Foo()
{
    union
    {
        int32_t Int;
        float Float;
    };

    // Int and Float are added to Foo, so they can be used directly
    Float = 3.14f;
    DebugLog(Int); // 1078523331
}
```

This feature is commonly used to create what's called a "tagged union" by wrapping the union and an enum in a struct:

```
struct IntOrFloat
{
    // The "tag" remembers the active member
    enum { Int, Float } Type;

    // Anonymous union
    union
    {
        int32_t IntVal;
        float FloatVal;
    };
};

IntOrFloat iof;

iof.FloatVal = 3.14f; // Set value
iof.Type = IntOrFloat::Float; // Set type

// Read value and type
DebugLog(iof.IntVal, iof.Type); // 1078523331, Float
```

This pattern is also called a "variant," typically when more protections are added to ensure the type and value are linked:

```
struct TypeException
{
};

class IntOrFloat
{
public:

    enum struct Type { Int, Float };

    Type GetType() const
    {
        return Type;
    }

    void SetIntVal(int32_t val)
    {
        Type = Type::Int;
        IntVal = val;
    }

    int32_t GetIntVal() const
    {
        if (Type != Type::Int)
        {
            throw TypeException{};
        }
        return IntVal;
```

```
    }

    void SetFloatVal(float val)
    {
        Type = Type::Float;
        FloatVal = val;
    }

    float GetFloatVal() const
    {
        if (Type != Type::Float)
        {
            throw TypeException{};
        }
        return FloatVal;
    }

private:

    Type Type;

    union
    {
        int32_t IntVal;
        float FloatVal;
    };
};
```

```
IntOrFloat iof;
iof.SetFloatVal(3.14f); // Set value to 3.14f and type to
Float
DebugLog(iof.GetFloatVal()); // 3.14
DebugLog(iof.GetIntVal()); // Throws exception: type is
not Int
```

Another common use of unions is to provide an alternative access mechanism without changing the type of the data. It's very common to see vectors, matrices, and quaternions that use unions to provide either named field access *or* array access to the components:

```
union Vector2
{
    struct
    {
        float X;
        float Y;
    };

    float Components[2];
};


Vector2 v;

// Named field access
v.X = 2;
```

```
v.Y = 4;

// Array access: same values due to union
DebugLog(v.Components[0], v.Components[1]); // 2, 4

// Array access
v.Components[0] = 20;
v.Components[1] = 40;

// Named field access: same values due to union
DebugLog(v.X, v.Y); // 20, 40
```

Pointers to Members

Finally, let's look at how we create pointers to members of structs. To simply get a pointer to a specific struct instance's non-static data member, we can use the normal pointer syntax:

```
struct Vector2
{
    float X;
    float Y;
};

Vector2 v{2, 4};
float* p = &v.X; // p points to the X data member of a
```

However, we can also get a pointer to a non-static data member of *any* instance of the struct:

```
float Vector2::* p = &Vector2::X; // p points to the X
data member of a Vector2
```

To dereference such a pointer, we need an instance of the struct whose data member it points at:

```
float Vector2::* p = &Vector2::X;
Vector2 v{2, 4};
```

```
// Dereference the pointer for a particular struct
DebugLog(v.*p); // 2
```

These pointers can't be converted to plain pointers or visa versa, but polymorphism is allowed as long as the base class isn't `virtual`:

```
struct Vector2
{
    float X;
    float Y;
};

struct Vector3 : Vector2
{
    float Z;
};

float Vector2::* p = &Vector2::X;
Vector2 v{2, 4};

float* p2 = p; // Compiler error: not compatible

float f = 3.14f;
float Vector2::* pf = &f; // Compiler error: not
compatible

float Vector3::* p3 = p; // OK: Vector3 derives from
```

```
Vector2
DebugLog(v.*p3); // 2
```

The syntax gets a little complicated when making a pointer to a member that is itself a pointer to a member. Thankfully, this is rarely seen:

```
struct Float
{
    float Val;
};

struct PtrToFloat
{
    float Float::* Ptr;
};

// Pointer to Val in a Float pointed to by Ptr in a
PtrToFloat
float Float::* PtrToFloat::* p1 = &PtrToFloat::Ptr;

Float f{3.14f};
PtrToFloat ptf{&Float::Val};

float Float::* pf = ptf.*p1; // Dereference first level
of indirection
float floatVal = f.*pf; // Dereference second level of
indirection
```

```
DebugLog(floatVal); // 3.14

// Dereference both levels of indirection at once
DebugLog(f.*(ptf.*p1)); // 3.14
```

Pointers to member functions can also be taken. The syntax is like a combination of data member pointers and normal function pointers:

```
struct Player
{
    int32_t Health;
};

struct PlayerOps
{
    Player& Target;

    PlayerOps(Player& target)
        : Target(target)
    {
    }

    void Damage(int32_t amount)
    {
        Target.Health -= amount;
    }

    void Heal(int32_t amount)
```

```
    {
        Target.Health += amount;
    }
};

// Pointer to a non-static member function of PlayerOps
that
// takes an int32_t and returns void
void (PlayerOps::* op)(int32_t) = &PlayerOps::Damage;

Player player{100};
PlayerOps ops(player);

// Call the Damage function via the pointer
(ops.*op)(20);
DebugLog(player.Health); // 80

// Re-assign to another compatible function
op = &PlayerOps::Heal;

// Call the Heal function via the pointer
(ops.*op)(10);
DebugLog(player.Health); // 90
```

Conclusion

This chapter we've seen a bunch of miscellaneous class functionality that isn't available in C#. User-defined literals can make code both more expressive and more terse at the same time. It's best used sparingly for very stable, core types like the Standard Library's `string`.

Local classes give a lot of the same benefits that local functions do in C#, but go a step further and allow nearly full class functionality including data members, constructors, destructors, and overloaded operators.

Copy and move assignment operators allow us to easily copy and move classes with the familiar `x = y` syntax rather than utility functions typically named `Clone` or `Copy`. The compiler will even generate them for us, saving a lot of boilerplate and potential for errors if that boilerplate gets out of sync with changes to the class.

Unions allow for memory savings, advanced manipulation of the bits and bytes behind types like `float`, and the convenience of alternative access styles. They can be partially emulated in C#, but native support in C++ is more convenient and offers more advanced functionality.

Pointers to members allow us to limit them to pointing specifically to members of classes and to not tie that access to any particular instance of the class. With support for both data members and member functions, we have a tool that enables runtime determination of the data to use or function to call without needing a heavyweight language feature like C#'s delegates. This can be used for setting modes (e.g. `Damage` mode versus `Heal` mode), for GUI callbacks (e.g. click handlers), or a variety of other situations.

17. Namespaces

Namespace Basics

Namespaces serve the same high-level purpose in C++ as they do in C#. They allow us to reuse identifiers and disambiguate them by namespace. The basic syntax even looks the same:

```
namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };
}
```

Also like C#, we can reopen the namespace to add on to it by simply reusing the name:

```
namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };
}
```

```
namespace Math
{
    struct Vector3
    {
        float X;
        float Y;
        float Z;
    };
}
```

We can also nest namespaces:

```
namespace Math
{
    namespace LinearAlgebra
    {
        struct Vector2
        {
            float X;
            float Y;
        };
    }
}
```

Accessing members of the namespace is a bit different. As we've seen with enums and structs, we continue to use the scope resolution operator (`A::B`) instead of C#'s dot syntax (`A.B`).

```
Math::Vector2 vec{2, 4}; // Refer to Vector2 in the Math
namespace
DebugLog(vec.X, vec.Y); // 2, 4
```

To refer to the namespace implicitly created for the global scope, we use `::B` instead of `global::B` as in C#:

```
int32_t highScore = 0;

class Player
{
    int32_t numPoints;
    int32_t highScore;

    void ScorePoints(int32_t num)
    {
        numPoints += num;

        // highScore refers to the data member
        if (numPoints > highScore)
        {
            highScore = numPoints;
        }

        // ::highScore refers to the global variable
        if (numPoints > ::highScore)
        {
```

```
            ::highScore = numPoints;
        }
    }
};
```

As of C++17, we can also use the scope resolution operator to create nested namespaces:

```
namespace Math::LinearAlgebra
{
    struct Vector2
    {
        float X;
        float Y;
    };
}
```

Unlike C#, we're not limited to only putting types like structs and enums in a namespace. We can put anything we want there:

```
namespace Math
{
    // Variable
    const float PI = 3.14f;

    // Function
    bool IsNearlyZero(float val, float threshold=0.0001f)
    {
```

```
        return abs(val) < threshold;
    }
}
```

We can also put declarations inside the namespace and definitions outside:

```
namespace Math
{
    // Declarations
    struct Vector2;
    bool IsNearlyZero(float val, float
threshold=0.0001f);
}

// Definitions
struct Math::Vector2
{
    float X;
    float Y;
};
bool Math::IsNearlyZero(float val, float threshold)
{
    return abs(val) < threshold;
}
```

The definitions need to be in either an enclosing namespace or at global scope:

```
namespace Math
{
    // Declarations
    struct Vector2;
    bool IsNearlyZero(float val, float
threshold=0.0001f);
}

// Definitions
namespace Other
{
    // Compiler error: Other isn't an enclosing namespace
or global scope
    struct Math::Vector2
    {
        float X;
        float Y;
    };

    // Compiler error: Other isn't an enclosing namespace
or global scope
    bool Math::IsNearlyZero(float val, float threshold)
    {
        return abs(val) < threshold;
    }
}
```

Note that a namespace with only functions is another way to mimic a C# `static class`.

Using Directives

Explicitly writing out namespace names like `Math::` gets tedious and verbose. As in C#, C++ has `using` directives to alleviate this issue. The syntax looks similar:

```
// Using directive
using namespace Math;

// No need for Math::
Vector2 vec{2, 4};
DebugLog(vec.X, vec.Y); // 2, 4
```

Unlike C# where `using` directives have to be at the top of a file, C++ allows them anywhere in the global scope, in a namespace, or even in any block:

```
namespace MathUtils
{
    // Using directive inside a namespace
    using namespace Math;

    bool IsNearlyZero(Vector2 vec, float
threshold=0.0001f)
    {
        return abs(vec.X) < threshold && abs(vec.Y) <
threshold;
    }
```

```
}

void Foo()
{
    // Using directive inside a function
    using namespace Math;

    // No need for Math::
    Vector2 vec{2, 4};
    DebugLog(vec.X, vec.Y); // 2, 4
}

enum struct Op
{
    IS_NEARLY_ZERO
};

bool DoOp(Math::Vector2 vec, Op op)
{
    if (op == Op::IS_NEARLY_ZERO)
    {
        // Using directive inside a block
        using namespace MathUtils;

        return IsNearlyZero(vec);
    }
    return false;
}
```

Also unlike C#, `using` directives are transitive. In the above, the `MathUtils` namespace has `using namespace Math`. That means any `using namespace MathUtils` implicitly includes a `using namespace Math`:

```
// Implicitly includes MathUtils' "using namespace Math"
using namespace MathUtils;

// No need for Math:: due to transitive using directive
Vector2 vec{2, 4};

// No need for MathUtils::
DebugLog(IsNearlyZero(vec)); // false
```

Even members added to the namespace *after* the `using` directive are included transitively:

```
namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };
}

namespace MathUtils
{
```

```
    using namespace Math;

    bool IsNearlyZero(Vector2 vec, float
threshold=0.0001f)
    {
        return abs(vec.X) < threshold && abs(vec.Y) <
threshold;
    }
}

namespace Math
{
    struct Vector3
    {
        float X;
        float Y;
        float Z;
    };
}

void Foo()
{
    // Implicitly includes MathUtils' "using namespace
Math"
    // Includes Vector3, even though it was after "using
namespace Math"
    using namespace MathUtils;
```

```
    Vector3 vec{2, 4, 6};
    DebugLog(vec.X, vec.Y, vec.Z); // 2, 4, 6
}
```

Note that it is generally considered a bad practice to place `using` directives in the global scope of header files as it imposes the entirety of the namespace as well as any transitively-used namespaces on all files that `#include` it.

Inline Namespaces

C++ namespaces may be `inline`:

```
inline namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };
}
```

This is similar to a non-`inline` namespace immediately followed by a `using` directive:

```
namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };
}
using namespace Math;
```

We can therefore use the members of the namespace without the scope resolution operator or an explicit `using` directive:

```
Vector2 vec{2, 4};
DebugLog(vec.X, vec.Y); // 2, 4
```

As of C++20, we can add the keyword `inline` before each name except the first when defining nested namespaces:

```
// Math is a non-inline namespace
// LinearAlgebra is an inline namespace nested in Math
namespace Math::inline LinearAlgebra
{
    struct Vector2
    {
        float X;
        float Y;
    };
}

// Math:: still required as Math is not an inline
namespace
Math::Vector2 vec{2, 4};
DebugLog(vec.X, vec.Y); // 2, 4
```

This adds convenience to a normal use case for `inline` namespaces. It's typical to want to group together functionality, but

also offer it piecemeal. For example, the C++ Standard Library offers user-defined literals for the `string` and `hour` types like this:

```
namespace std::inline literals::inline string_literals
{
    std::string operator""s(const char* chars, size_t
len)
    {
        // ... implementation ...
    }
}
namespace std:inline literals::inline chrono_literals
{
    std::chrono::hours operator""h(long double val)
    {
        // ... implementation ...
    }
}

void UseAllLiterals()
{
    // Transitively use string_literals and
chrono_literals
    // No need to specify each one
    using namespace std::literals;

    std::string greeting = "hello"s;
    std::chrono::hours halfHour = 0.5h;
```

```
}

void UseJustStringLiterals()
{
    // Use just string_literals
    using namespace std::literals::string_literals;

    std::string greeting = "hello"s;
    std::chrono::hours halfHour = 0.5h; // Compiler error
}

void UseJustChronoLiterals()
{
    // Use just chrono_literals
    using namespace std::literals::chrono_literals;

    std::string greeting = "hello"_s; // Compiler error
    std::chrono::hours halfHour = 0.5h;
}
```

Unnamed Namespaces

Namespaces may have no name. Just like with `inline` namespaces, these have an implicit `using` directive right after them:

```
namespace
{
    struct Vector2
    {
        float X;
        float Y;
    };
}

// Implicitly added by the compiler
// UNNAMED is just a placeholder for the name the
compiler gives the namespace
using namespace UNNAMED;

// Can use members of the unnamed namespace
Vector2 vec{2, 4};
DebugLog(vec.X, vec.Y); // 2, 4
```

Because these namespaces have no name, there's no way to explicitly refer to their members with the scope resolution operator (`A::B`) or name them in a `using` directive.

Specially, all members of unnamed namespaces, including nested namespaces, have internal linkage, just like `static` global variables.

```
// other.cpp
int32_t Global; // External linkage
namespace
{
    int32_t InNamespace; // Internal linkage
}

// test.cpp
extern int32_t Global; // OK: has external linkage
extern int32_t InNamespace; // Linker error: has internal
linkage
                            // Also, no way to name the
namespace here
void Foo()
{
    Global = 123;
    InNamespace = 456;
}
```

Using Declarations

Besides `using` directives, C++ also has `using` declarations:

```
namespace Math
{
    struct Vector2
    {
        float X;
        float Y;
    };

    struct Vector3
    {
        float X;
        float Y;
        float Z;
    };
}

// Use just Vector2, not Vector3
using Math::Vector2;

Vector2 vec2{2, 4}; // OK
Vector3 vec3a{2, 4, 6}; // Compiler error
Math::Vector3 vec3b{2, 4, 6}; // OK
```

Like variable declarations, we can name multiple namespace members in a single `using` declaration:

```
namespace Game
{
    class Player;
    class Enemy;
}

void Foo()
{
    // Use Vector2 and Player, not Vector3 or Enemy
    using Alias::Vector2, Game::Player;

    Vector2 vec2{2, 4}; // OK
    Vector3 vec3{2, 4}; // Compiler error
    Player* player; // OK
    Enemy* enemy; // Compiler error
}
```

Unlike `using` directives, `using` declarations actually add the specified namespace members to the block they're declared in. This means some conflicts are possible:

```
namespace Stats
{
    int32_t score;
}
```

```
namespace Game
{
    struct Player
    {
        int32_t Score;
    };
}

bool HasHighScore(Game::Player* player)
{
    using Stats::score;

    int32_t score = player->Score; // Compiler error:
score already declared
    return score > score; // Ambiguous reference to score
}
```

There's one case where it's OK to have more than one identifier with the same name: function overloads. With multiple `using` declarations referring to functions with the same name, we can create an overload set within the function:

```
namespace Game
{
    struct Player
    {
        int32_t Health;
```

```
    };
}

namespace Damage
{
    struct Weapon
    {
        int32_t Damage;
    };

    void Use(Weapon& weapon, Game::Player& player)
    {
        player.Health -= weapon.Damage;
    }
}

namespace Healing
{
    struct Potion
    {
        int32_t HealAmount;
    };

    void Use(Potion& potion, Game::Player& player)
    {
        player.Health += potion.HealAmount;
    }
}
```

```
void DamageThenHeal(
    Game::Player& player, Damage::Weapon& weapon,
Healing::Potion& potion)
{
    using Damage::Use; // Now have one Use function
    using Healing::Use; // Now have two Use functions: an
overload set

    Use(weapon, player); // Call the Use(Weapon&,
Player&) overload
    Use(potion, player); // Call the Use(Potion&,
Player&) overload
}

Game::Player player{100};
Damage::Weapon weapon{20};
Healing::Potion potion{10};
DamageThenHeal(player, weapon, potion);
DebugLog(player.Health); // 90
```

Namespace Aliases

The final namespace feature is a simple one: aliases. We can use these to shorten long, usually nested namespace names:

```
namespace Math
{
    namespace LinearAlgebra
    {
        struct Vector2
        {
            float X;
            float Y;
        };
    }
}

// mla is an alias for Math::LinearAlgebra
namespace mla = Math::LinearAlgebra;

mla::Vector2 vec2{2, 4};
DebugLog(vec2.X, vec2.Y); // 2, 4
```

This is pretty close to `using N1 = N2` in C#. The major difference is that it can be placed in the global scope, in a namespace, or in any other block. It doesn't need to be placed at the top of the file and only apply to that one file.

Conclusion

C++ namespaces are, in many ways, very similar to C# namespaces. As is typical, they form a rough superset of the C# functionality. Advanced features like `inline` namespaces, `using` declarations, and unnamed namespaces are available to us. We also have the ability to put variables and functions in them in addition to type definitions. We can even put `using` declarations and directives nearly anywhere, not just at the top of files.

As is typical with more power though, more complexity is involved. We need to avoid overly-broad `using` directives, pay attention to transitive `using` directives, and resolve identifier conflicts with `using` declarations.

Namespaces are used in nearly every C++ codebase, as they are in nearly every C# codebase. We now know the rules for how to use them, so we're one step closer to effectively writing C++!

18. Exceptions

Throwing Exceptions

The syntax for throwing an exception looks almost the same in C++ as it does in C#:

```
throw e;
```

The major difference between the two languages is that C# requires exception objects to be class instances derived from `System.Exception`. There is a `std::exception` class in the C++ Standard Library, but we're free to ignore it and throw objects of any type: class instances, enums, primitives, pointers, etc.

```
class IOException {};
enum class ErrorCode { FileNotFound };

void Foo()
{
    // Class instance
    throw IOException{};

    // Enum
    throw ErrorCode::FileNotFound;

    // Primitive
    throw 1;
}
```

Note that the class instance of `IOException` here isn't a pointer or reference to an instance of the class. That's required by C# as all class instance variables are managed references. Here we throw the object itself, but we can also throw pointers if we want:

```
IOException ex;

void Foo()
{
    // Pointer to a class instance
    throw &ex;
}
```

It's typical to see `throw` all by itself in a statement but, like in C# 7.0, it's actually an expression that can be part of more complex statements. Here it is as commonly seen with the ternary/conditional operator:

```
class InvalidId{};
const int32_t MAX_PLAYERS = 4;
int32_t highScores[MAX_PLAYERS]{};

int32_t GetHighScore(int32_t playerId)
{
    return playerId < 0 || playerId >= MAX_PLAYERS ?
        throw InvalidId{} :
        highScores[playerId];
}
```

Catching Exceptions

Exceptions are caught with `try` and `catch` blocks, just like in C#:

```
void Foo()
{
    const int32_t id = 4;
    try
    {
        GetHighScore(id);
    }
    catch (InvalidId)
    {
        DebugLog("Invalid ID", id);
    }
}
```

We didn't give the caught exception object a name in this example because the mere fact that it was thrown is sufficient for the code in the `catch` block. That's also allowed in C#, as is naming the caught exception:

```
struct InvalidId
{
    int32_t Id;
};
const int32_t MAX_PLAYERS = 4;
int32_t highScores[MAX_PLAYERS]{};
```

```
int32_t GetHighScore(int32_t playerId)
{
    return playerId < 0 || playerId >= MAX_PLAYERS ?
        throw InvalidId{playerId} :
        highScores[playerId];
}

void Foo()
{
    try
    {
        GetHighScore(4);
    }
    catch (InvalidId ex)
    {
        DebugLog("Invalid ID", ex.Id);
    }
}
```

In this version, `InvalidId` has the ID that was invalid so we give the `catch` block's exception object a name in order to access it.

Catching multiple types of exception objects also looks the same as C#:

```
struct InvalidId
{
    int32_t Id;
```

```
};
struct NoHighScore
{
    int32_t PlayerId;
};
const int32_t MAX_PLAYERS = 4;
int32_t highScores[MAX_PLAYERS]{-1, -1, -1, -1};

int32_t GetHighScore(int32_t playerId)
{
    if (playerId < 0 || playerId >= MAX_PLAYERS)
    {
        throw InvalidId{playerId};
    }
    const int32_t highScore = highScores[playerId];
    return highScore < 0 ? throw NoHighScore{playerId} :
highScore;
}

void Foo()
{
    try
    {
        GetHighScore(2);
    }
    catch (InvalidId ex)
    {
        DebugLog("Invalid ID", ex.Id);
```

```
    }
    catch (NoHighScore ex)
    {
        DebugLog("No high score for player with ID",
ex.PlayerId);
    }
}
```

The `catch` blocks are checked in the order they're listed and the first matching type's `catch` block gets executed.

Catching all types of exceptions looks a bit different in C++. We use `catch (...) {}` instead of just `catch {}`:

```
void Foo()
{
    try
    {
        GetHighScore(2);
    }
    catch (...)
    {
        DebugLog("Couldn't get high score");
    }
}
```

As in C#, we can re-throw caught exceptions, whether they have a name or not, using `throw;`:

```
void Foo()
{
    try
    {
        GetHighScore(2);
    }
    catch (...)
    {
        throw;
    }
}
```

C# has an exception filters feature:

```
catch (Exception e) when (e.Message == "kaboom")
{
    Console.WriteLine("kaboom!");
}
catch (Exception e) when (e.Message == "boom")
{
    Console.WriteLine("boom!");
}
```

This isn't available in C++, but we can approximate it with regular code such as a `switch`. Just keep in mind that C# exception filters are evaluated before stack unwinding and this approximation is evaluated afterward:

```
enum class IOError { FileNotFound, PermissionDenied };

void Foo()
{
    try
    {
        DeleteFile("/path/to/file");
    }
    catch (IOError err)
    {
        switch (err)
        {
            case IOError::FileNotFound:
                DebugLog("file not found");
                break;
            case IOError::PermissionDenied:
                DebugLog("permission denied");
                break;
            default:
                throw;
        }
    }
}
```

Lastly, C++ has an alterate form of `try`–`catch` blocks that are placed at the function level:

```
void Foo() try
{
    GetHighScore(2);
}
catch (...)
{
    DebugLog("Couldn't get high score");
}
```

These are similar to a `try` that encompasses the whole function. The main reason to use one is to be able to catch exceptions in constructor initializer lists. Since these don't appear in the function body, there's no other way to write a `try` block that includes them.

At the time the function-level `catch` block is called, all constructed data members have already been destroyed. At the end of the `catch`, the exception is automatically re-thrown with an implicit `throw;` similar to the implicit `return;` at the of a `void` function:

```
struct HighScore
{
    int32_t Value;

    HighScore(int32_t playerId) try
        : Value(GetHighScore(playerId))
    {
    }
    catch (...)
    {
```

```
        DebugLog("Couldn't get high score");
    }
};

void Foo()
{
    try
    {
        HighScore hs{2};
    }
    catch (NoHighScore ex)
    {
        DebugLog("No high score for player",
ex.PlayerId);
    }
}

// This prints:
// * Couldn't get high score
// * No high score for player 2
```

Parameters, but not local variables, can be used in a function-level `catch` block and they may even `return`:

```
int32_t GetHighScoreOrDefault(int32_t playerId, int32_t
defaultVal) try
{
    return GetHighScore(playerId);
```

```
}
catch (...)
{
    DebugLog(
        "Couldn't get high score for", playerId,
        ". Returning default value", defaultVal);
    return defaultVal;
}

void Foo()
{
    DebugLog(GetHighScoreOrDefault(2, -1));
}

// This prints:
// * Couldn't get high score for 2. Returning default
value -1
// * -1
```

Exception Specifications

C++ functions are classified as either non-throwing or potentially-throwing. By default, all functions are potentially-throwing except destructors and compiler-generated functions that don't call potentially-throwing functions.

```
// Regular function is potentially-throwing
void Foo() {}

struct MyStruct
{
    // Compiler-generated constructor is non-throwing
    //MyStruct()
    //{
    //}

    // Destructor is non-throwing
    ~MyStruct()
    {
    }
};
```

This information is used by the compiler to produce more optimized code and to enable compile-time checks in case we accidentally throw in a non-throwing function.

We can override the default classification in two ways. First, by adding `noexcept` after the function's parameter list like where we put `override` or `const`:

```
void Foo() noexcept // Force non-throwing
{
    throw 1; // Compiler warning: throwing in a non-
throwing function
}
```

We can make `noexcept` conditional by adding a compile-time expression in parentheses after it:

```
void Foo() noexcept(FOO_THROWS == 1)
{
    throw 1;
}
```

Compiler options such as `-DFOO_THROWS=1` can be used to set `FOO_THROWS` to change the function's throwing classification without changing the code.

We can also add `noexcept` to function pointers in the same way:

```
void Foo() noexcept
{
    throw 1;
}

void Goo() noexcept(FOO_THROWS == 1)
{
    throw 1;
```

```
}

void (*pFoo)() noexcept = Foo;
void (*pGoo)() noexcept(FOO_THROWS == 1) = Goo;
```

The second way of changing the default was deprecated in C++11 and removed completely in C++17 and C++20. It used to specify the types of exceptions that a function could throw or that a function wouldn't throw any exceptions at all:

```
// Force non-throwing
// Deprecated in C++11 and removed in C++20
void Foo() throw()
{
    throw 1; // Compiler warning: this function is non-
throwing
}

// Can throw an int or a float
// Deprecated in C++11 and removed in C++17
void Goo(int a) throw(int, float)
{
    if (a == 1)
    {
        throw 123; // Throw an int
    }
    else if (a == 2)
    {
```

```
            throw 3.14f; // Throw a float
        }
    }
```

Stack Unwinding

Just like when we throw exceptions in C#, exceptions thrown in C++ unwind the call stack looking for a `try` block that can handle the exception. This triggers `finally` blocks in C#, but C++ doesn't have `finally` blocks. Instead, destructors of local variables are called without the need for any explicit syntax such as `finally`:

```
struct File
{
    FILE* handle;

    File(const char* path)
    {
        handle = fopen(path, "r");
    }

    ~File()
    {
        fclose(handle);
    }

    void Write(int32_t val)
    {
        fwrite(&val, sizeof(val), 1, handle);
    }
};
```

```
void Foo()
{
    File file{"/path/to/file"};
    int32_t highScore = GetHighScore(123);
    file.Write(highScore);
}
```

If `GetHighScore` throws an exception in this example, the destructor of `file` will be called and the file handle will be closed and relinquished to the OS. If `GetHighScore` doesn't throw an exception, the lifetime of `file` will come to an end at the end of the function and its destructor will be called. In either case, a resource leak is prevented and no `try` or `finally` block needs to be written.

As the call stack is unwound looking for suitable `catch` blocks, we may reach the root function of the call stack and still not have caught the exception. In this case, the C++ Standard Library function `std::terminate` is called. This calls the `std::terminate_handler` function pointer. It defaults to a function that calls `std::abort`, which effectively crashes the program. We can set our own `std::terminate_handler` function pointer, typically to perform some kind of crash reporting before calling `std::abort`:

```
void SaveCrashReport()
{
    // ...
}

void OnTerminate()
{
```

```
    SaveCrashReport();
    std::abort();
}

std::set_terminate(OnTerminate);

// ... anywhere else in the program ...

throw 123; // calls OnTerminate if not caught
```

`std::terminate` is also called in many other circumstances. One of these is if a destructor called during stack unwinding itself throws an exception:

```
struct Boom
{
    ~Boom() noexcept(false) // Force potentially-throwing
    {
        DebugLog("boom!");

        // If called during stack unwinding, this calls
std::terminate
        // Otherwise, it just throws like normal
        throw 123;
    }
};

void Foo()
```

```
{
    try
    {
        Boom boom{};
        throw 456; // Calls boom's destructor
    }
    catch (...)
    {
        DebugLog("never printed");
    }
}
```

Another way `std::terminate` could be called is if a non-throwing function throws:

```
struct Boom
{
    ~Boom() // Non-throwing
    {
        throw 123; // Compiler warning: throwing in non-throwing function
    }
};

void Foo()
{
    try
    {
```

```
        Boom boom{};
    }
    catch (...)
    {
        DebugLog("never printed");
    }
}
```

Or if a `static` variable's constructor throws an exception:

```
struct Boom
{
    Boom()
    {
        throw 123;
    }
};

static Boom boom{};
```

Note that throwing in a `static` local variables' constructor doesn't call `std::terminate`. Instead, the constructor is just called again the next time the function is called:

```
struct Boom
{
    Boom()
    {
```

```
        throw 123;
    }
};

void Goo()
{
    static Boom boom{}; // Static local variable who's
constructor throws
}

void Foo()
{
    for (int i = 0; i < 3; ++i)
    {
        try
        {
            Goo();
        }
        catch (...)
        {
            DebugLog("caught"); // Prints three times
        }
    }
}
```

Slicing

One common mistake is to catch class instances that are part of an inheritance hierarchy. We typically want to catch the base class (`IOError`) to implicitly catch all the derived classes (`FileNotFound`, `PermissionDenied`). This will lead to "slicing" off the base class sub-object of the derived class. Since the subobject is really designed to be used as a part of the derived class object, this may cause errors.

To see this in action, consider the following case where `virtual` functions aren't respected:

```
struct Exception
{
    const char* Message;

    virtual void Print()
    {
        DebugLog(Message);
    }
};

struct IOException : Exception
{
    const char* Path;

    IOException(const char* path, const char* message)
    {
        Path = path;
```

```
        Message = message;
    }

    virtual void Print() override
    {
        DebugLog(Message, Path);
    }
};

FILE* OpenLogFile(const char* path)
{
    FILE* handle = fopen(path, "r");
    return handle == nullptr ? throw IOException{path,
"Can't open"} : handle;
}

void Foo()
{
    try
    {
        FILE* handle = OpenLogFile("/path/to/log/file");
        // ... use handle
    }
    // Catching the base class slices it off from the
whole IOException
    catch (Exception ex)
    {
        // Calls Exception's Print, not IOException's
```

```
Print
        ex.Print();
    }
}
```

To fix the issue, simply catch by reference:

```
catch (Exception& ex)
```

A `const` reference is usually even better since it's rare to want to modify the exception object:

```
catch (const Exception& ex)
```

Either way, the appropriate `virtual` function will now be called and we'll get the right error message:

```
Can't open /path/to/log/file
```

Conclusion

Exceptions in C++ are broadly similar to exceptions in C#. Both languages can throw class instances and catch one, many, or arbitrary types. C++ lacks `catch` filters, but emulates it with normal code like `switch` statements. It lacks `finally` because destructors are used instead without the need to remember to add `try` or `finally` blocks.

C++ also gains the ability to throw objects, not just references. Those objects don't have to be class instances as primitives, enums, and pointers are also allowed. We can also gain a compiler safety net and better optimization by using `noexcept` specifications. When exceptions go uncaught, we can hook into `std::terminate_handler` to add crash reporting or take any other actions before the program exits.

19. Dynamic Allocation

History and Strategy

Let's start by looking at a bit of a history which is still very relevant to C++ programming today. In C, not C++, memory is dynamically allocated using a family of functions in the C Standard Library whose names end in `alloc`:

```
// Dynamically allocate 4 bytes
void* memory = malloc(4);

// Check for allocation failure
// Not necessary for small (e.g. 4 byte) allocations
// Needed for large (e.g. array) allocations
if (memory != NULL)
{
    // Cast to treat it as a pointer to an int
    int* pInt = (int*)memory;

    // Read the memory
    // This is undefined behavior: the memory hasn't been
initialized!
    DebugLog(*pInt);

    // Release the memory
    free(memory);

    // Write the memory
```

```
    // This is undefined behavior: the memory has been
released!
    *pInt = 123;

    // Release the memory again
    // This is undefined behavior: the memory has already
been released!
    free(memory);
}
```

“Raw” use of `malloc` and `free` like this is still common in C++ codebases. It’s a pretty low-level way of working though, and generally discouraged in most C++ codebases. That’s because it’s quite easy to accidentally trigger undefined behavior. The three mistakes in the above code are very common bugs.

Higher-level dynamic allocation approaches make these mistakes either harder to make or impossible. For example, in C# there’s no way to get memory that hasn’t been initialized since everything is set to zero, no way to have a reference to released memory since it’s only released after the last reference is relinquished, and no way to double-release memory since that’s handled by the GC.

C++ doesn’t take such a high-level approach as C# since the above C code is also legal C++. It does, however, provide many higher-level facilities for the majority of cases where safety is preferable to total control.

Allocation

The `new` operator in C++ is conceptually similar to using the `new` operator with classes in C#. It dynamically allocates memory, initializes it, and evaluates a pointer:

```
struct Vector2
{
    float X;
    float Y;

    Vector2()
        : X(0), Y(0)
    {
    }

    Vector2(float x, float y)
        : X(x), Y(y)
    {
    }
};

// 1) Allocate enough memory for a Vector2:
sizeof(Vector2)
// 2) Call the constructor
//    * "this" is the allocated memory
//    * Pass 2 and 4 as arguments
// 3) Evaluate to a Vector2*
```

```
Vector2* pVec = new Vector2{2, 4};

DebugLog(pVec->X, pVec->Y); // 2, 4
```

The `new` operator combines several of the manual steps from the C code so we can't forget to do them or accidentally do them wrong. As a result, safety is increased in numerous ways:

- The amount of memory allocated is computed by the compiler, so it's always correct
- The allocated memory is always initialized (i.e. by the constructor), so we can't use it before it's initialized
- The initialization code is always passed the right pointer to the allocated memory
- The allocated memory is always cast to the correct type of pointer
- Allocation failures are always handled (more below)

C# allows us to use `new` with classes and structs. In C++, we can use `new` with any type:

```
// Dynamically allocate a primitive
int* pInt = new int{123};
DebugLog(*pInt); // 123

// Dynamically allocate an enum
enum class Color { Red, Green, Blue };
Color* pColor = new Color{Color::Green};
DebugLog((int)*pColor); // Red
```

We can also allocate arrays of objects with `new`. Just like other arrays, each element is initialized:

```
// Dynamically allocate an array of three Vector2
// The default constructor is called on each of them
Vector2* vectors = new Vector2[3]();

DebugLog(vectors[0].X, vectors[0].Y); // 0, 0
DebugLog(vectors[1].X, vectors[1].Y); // 0, 0
DebugLog(vectors[2].X, vectors[2].Y); // 0, 0
```

This is different from an array in C# in two ways. First, it's not a managed object as C++ doesn't have a garbage collector. Second, it's an array of `Vector2` *objects*, not *references* to `Vector2` objects. It's more like an array of C# structs than an array of C# classes: the objects are laid out sequentially in memory.

Initialization

Regardless of the type, `new` always takes the same steps: allocate, initialize, then evaluate to a pointer. Initialization is controlled by what we put after the name of the type: nothing, parentheses, or curly braces. If we put nothing, the object or array of objects is default-initialized:

```
// Calls default constructor for classes
Vector2* pVec1 = new Vector2;
DebugLog(pVec1->X, pVec1->Y); // 0, 0

// Does nothing for primitives
int* pInt1 = new int;
DebugLog(*pInt1); // Undefined behavior: int not
initialized

// Calls default constructor for classes
Vector2* vectors1 = new Vector2[1];
DebugLog(vectors1[0].X, vectors1[0].Y); // 0, 0

// Does nothing for primitives
int* ints1 = new int[1];
DebugLog(ints1[0]); // Undefined behavior: ints not
initialized
```

If we put parentheses, a single object is direct-initialized:

```
// Calls (float, float) constructor
Vector2* pVec2 = new Vector2(2, 4);
DebugLog(pVec2->X, pVec2->Y); // 2, 4

// Sets to 123
int* pInt2 = new int(123);
DebugLog(*pInt2); // 123
```

Parentheses with an array must be empty. This aggregate-initializes the array:

```
// Calls default constructor for classes
Vector2* vectors2 = new Vector2[1]();
DebugLog(vectors2[0].X, vectors2[0].Y); // 0, 0

// Sets to zero
int* ints2 = new int[1]();
DebugLog(ints2[0]); // 0
```

Curly braces list-initialize single objects:

```
// Calls (float, float) constructor
Vector2* pVec3 = new Vector2{2, 4};
DebugLog(pVec3->X, pVec3->Y); // 2, 4

// Sets to 123
```

```
int* pInt3 = new int{123};
DebugLog(*pInt3); // 123
```

They aggregate-initialize arrays and can be non-empty, such as to pass arguments to a constructor or set primitives to a value. This is generally the recommended form for both arrays and single objects:

```
// Calls (float, float) constructor for each element
Vector2* vectors3 = new Vector2[1]{2, 4};
DebugLog(vectors3[0].X, vectors3[0].Y); // 2, 4

// Sets each element to 123
int* ints3 = new int[1]{123};
DebugLog(ints3[0]); // 123
```

When memory allocation fails, such as when there's not enough , `new` will throw a `std::bad_alloc` exception:

```
try
{
    // Attempt a 1 TB allocation
    // Throws an exception if the allocation fails
    char* big = new char[1024*1024*1024*1024];

    // Never executed if the allocation fails
    big[0] = 123;
}
catch (std::bad_alloc)
```

```
{
    // This gets printed if the allocation fails
    DebugLog("Failed to allocate big array");
}
```

Some codebases, especially in games, prefer to avoid exceptions. Compilers often provide an option to call `std::abort` to crash the program instead even though this is technically a violation of the C++ standard:

```
// Attempt a 1 TB allocation
// Calls abort() if the allocation fails
char* big = new char[1024*1024*1024*1024];

// Never executed if the allocation fails
big[0] = 123;
```

Deallocation

All of the above examples create memory leaks. That's because C++ has no garbage collector to automatically release memory that's no longer referenced. Instead, we must release the memory when we're done with it. We do that with the `delete` operator:

```
Vector2* pVec = new Vector2{2, 4};
DebugLog(pVec->X, pVec->Y); // 2, 4

// 1) Call the Vector2 destructor
// 2) Release the allocated memory pointed to by pVec
delete pVec;

DebugLog(pVec->X, pVec->Y); // Undefined behavior: the
memory has been released

delete pVec; // Undefined behavior: the memory has
already been released
```

The `delete` operator takes one more step toward safety by combining two steps together: de-initialization of the memory's contents followed by deallocating it. It doesn't, however, prevent the two errors at the end of the example: "use after release" and "double-release."

One way to address these issues is to set all pointers to the memory to null after releasing them:

```
delete pVec;
pVec = nullptr;

// Undefined behavior: derefrencing null
DebugLog(pVec->X, pVec->Y);

delete pVec; // OK
```

In the “use after release” case, our dereferencing of a null pointer is still undefined behavior. If the compiler can determine this, it can produce whatever machine code it wants. It may simply dereference null and crash, or it may do something strange like remove the `DebugLog` line completely.

Most of the time, such as when using the null pointer in some far-flung part of the codebase, the compiler can’t determine that it’s null and will assume a non-null pointer. In that case, dereferencing null will crash the program. So this is only a moderate improvement as we may only potentially get a crash instead of data corruption from reading or writing the released memory.

In the “double-release” case, it’s OK to `delete` null so this simply isn’t a problem anymore.

Because a `Vector2*` might be a pointer to a single `Vector2` or an array of `Vector2` objects, a second form of `delete` exists to call the destructors of all the elements in an array:

```
Vector2* pVectors1 = new Vector2[3]{Vector2{2, 4}};
// Correct:
// 1) Call the Vector2 destructor on all three vectors
```

```
// 2) Release the allocated memory pointed to by
pVectors1
delete [] pVectors1;

Vector2* pVectors2 = new Vector2[3]{Vector2{2, 4}};
// Bug:
// 1) Call the Vector2 destructor on THE FIRST vector
// 2) Release the allocated memory pointed to by
pVectors2
delete pVectors2;
```

Note that the correct destructor needs to be called, which can be problematic in the case of inheritance:

```
struct HasId
{
    int32_t Id;

    // Non-virtual destructor
    ~HasId()
    {
    }
};

struct Combatant
{
    // Non-virtual destructor
    ~Combatant()
```

```
    {
    }
};

struct Enemy : HasId, Combatant
{
    // Non-virtual destructor
    ~Enemy()
    {
    }
};

// Allocate an Enemy
Enemy* pEnemy = new Enemy();

// Polymorphism is allowed because Enemy "is a" Combatant
due to inheritance
Combatant* pCombatant = pEnemy;

// Deallocate a Combatant
// 1) Call the Combatant, not Enemy, destructor
// 2) Release the allocated memory pointed to by
pCombatant
delete pCombatant;
```

This is undefined behavior since the sub-object pointed to by `pCombatant` might not be the same as the pointer that was allocated. To fix this, use a `virtual` destructor:

```
struct HasId
{
    int32_t Id;

    virtual ~HasId()
    {
    }
};

struct Combatant
{
    virtual ~Combatant()
    {
    }
};

struct Enemy : HasId, Combatant
{
    virtual ~Enemy()
    {
    }
};

Enemy* pEnemy = new Enemy();
Combatant* pCombatant = pEnemy;

// Deallocate a Combatant
// 1) Call the Enemy destructor
```

```
// 2) Release the allocated memory pointed to by pEnemy
delete pCombatant;
```

Overloading New and Delete

So far we've been using the default `new` and `delete` operators. These are fine for most purposes, but sometimes we want to take more control over memory allocation and deallocation. For example, we might want to use an alternative allocator for improved performance as Unity's `Allocator.Temp` does in C#. To do this, we can overload the `new` and `delete` operators.

There are several forms the overloaded operators can take, but they should always be overloaded in pairs. Here's the simplest form:

```
// We need the std::size_t type
#include <cstddef>

struct Vector2
{
    float X;
    float Y;

    void* operator new(std::size_t count)
    {
        return malloc(sizeof(Vector2));
    }

    void operator delete(void* ptr)
    {
        free(ptr);
    }
```

```
};

// Calls overloaded new operator in Vector2
Vector2* pVec = new Vector2{2, 4};

DebugLog(pVec->X, pVec->Y); // 2, 4

// Calls overloaded delete operator in Vector2
delete pVec;
```

The array versions are overloaded separately:

```
struct Vector2
{
    float X;
    float Y;

    void* operator new[](std::size_t count)
    {
        return malloc(sizeof(Vector2)*count);
    }

    void operator delete[](void* ptr)
    {
        free(ptr);
    }
};
```

```
Vector2* pVecs = new Vector2[1];
delete [] pVecs;
```

Overloaded operators, including `new`, can take any arguments. We put them between the `new` keyword and the type to allocate:

```
struct Vector2
{
    float X;
    float Y;

    // Overload the new operator that takes (float, float) arguments
    void* operator new(std::size_t count, float x, float y)
    {
        // Note: for demonstration purposes only
        // Normal code would just use a constructor
        Vector2* pVec = (Vector2*)malloc(sizeof(Vector2)*count);
        pVec->X = x;
        pVec->Y = y;
        return pVec;
    }

    // Overload the normal delete operator
    void operator delete(void* memory, std::size_t count)
    {
```

```
        free(memory);
    }

    // Overload a delete operator corresponding with the
new operator
    // that takes (float, float) arguments
    void operator delete(void* memory, std::size_t count,
float x, float y)
    {
        // Forward the call to the normal delete operator
        Vector2::operator delete(memory, count);
    }
};

// Call the overloaded (float, float) new operator
Vector2* pVec = new (2, 4) Vector2;

DebugLog(pVec->X, pVec->Y); // 2, 4

// Call the normal delete operator
delete pVec;
```

One convention that's arisen is to take a `void*` as the second argument to indicate "placement new." In this case, no memory is allocated and the object simply uses the memory pointed to by that `void*`:

```
struct Vector2
{
    float X;
    float Y;

    // Overload the "placement new" operator
    // Mark "noexcept" because there's no way this can
throw
    void* operator new(std::size_t count, void* place)
noexcept
    {
        // Don't allocate. Just return the given memory
address.
        return place;
    }
};

// Allocate our own memory to hold the Vector2
// We can use global variables, the stack, or anything
else
char buf[sizeof(Vector2)];

// Call the "placement new" operator
// The Vector2 is put in buf
Vector2* pVec = new (buf) Vector2{2, 4};
DebugLog(pVec->X, pVec->Y); // 2, 4
```

```
// Note: no "delete" since we didn't actually allocate
memory
```

Like other overloaded operators, we can also overload outside the class to handle more than that one type. For example, here's a "placement new" for all types:

```
struct Vector2
{
    float X;
    float Y;
};

void* operator new(std::size_t count, void* place)
noexcept
{
    return place;
}

char buf[sizeof(Vector2)];
Vector2* pVec = new (buf) Vector2{2, 4};
DebugLog(pVec->X, pVec->Y); // 2, 4

float* pFloat = new (buf) float{3.14f};
DebugLog(*pFloat); // 3.14
```

Owning Types

So far we've overcome a lot of possible mistakes that could have been made with low-level dynamic allocation functions like `malloc` and `free`. Even so, "naked" use of `new` and `delete` is often frowned upon in "Modern C++" (i.e. C++11 and newer) codebases. This is because we are still susceptible to common bugs:

- Forgetting to call `delete`, resulting in a memory leak
- Calling `delete` twice, which is undefined behavior and likely a crash
- Using allocated memory after calling `delete`, which is undefined behavior and likely causes corruption

To alleviate these issues, `new` and `delete` operators are typically wrapped in a class referred to as an "owning type." This gives us access to constructors and destructors to allocate and deallocate memory much more safely. The C++ Standard Library has several generic types for this purpose which we'll cover later in the book. For now, let's build a simple "owning type" that owns an array of `float`:

```
class FloatArray
{
    int32_t length;
    float* floats;

public:

    FloatArray(int32_t length)
        : length{length}
```

```
        , floats{new float[length]{0}}
    {
    }

    float& operator[](int32_t index)
    {
        if (index < 0 || index >= length)
        {
            throw IndexOutOfBounds{};
        }
        return floats[index];
    }

    virtual ~FloatArray()
    {
        delete [] floats;
        floats = nullptr;
    }

    struct IndexOutOfBounds {};
};

try
{
    FloatArray floats{3};
    floats[0] = 3.14f;

    // Index out of bounds
```

```
    // Throws exception
    // FloatArray destructor called
    DebugLog(floats[-1]); // 3.14
}
catch (FloatArray::IndexOutOfBounds)
{
    DebugLog("whoops"); // Gets printed
}
```

Here we see that we've encapsulated the `new` or `delete` operators into the `FloatArray` class. The bulk of the codebase is simply a user of this class and it doesn't ever need to write a `new` or `delete` operator. Despite that, it's solved all three of the above problems:

- We can't forget to call `delete` because the destructor does, even if an exception is thrown
- We can't call `delete` twice because the destructor does this for us
- We can't use the memory after calling `delete` because we wouldn't have the variable to call member functions on

By using a class, we can also prevent other common errors:

- The constructor always initializes the elements of the array to avoid undefined behavior when reading them before writing them
- The overloaded array subscript (`[]`) operator performs bounds checks to avoid memory corruption

Still, this is a poor implementation of an “owning type” as it’s vulnerable to a variety of other problems. For example, the compiler generates a copy constructor which copies the `floats` pointer leading to a double-release:

```
void Foo()
{
    FloatArray f1{3};
    FloatArray f2{f1}; // Copies floats and length

    // 1) Call f1's destructor which deletes the
allocated memory
    // 2) Call f2's destructor which deletes the
allocated memory again: crash
}
```

Instead of creating custom owning types like `FloatArray`, it’s much more common to use a platform library class like `std::vector` in the C++ Standard Library or `TArray` in Unreal. The same goes for other owning types like `std::unique_ptr` and `std::shared_ptr`, the C++ Standard Library’s “smart pointers” to a single object.

Conclusion

C# provides very high-level memory management by requiring garbage collection. To avoid it, we're forced into "unsafe" code and must give up many language features including classes, interfaces, and delegates. Such is the case with Unity's Burst compiler, which impose the HPC# subset.

C++ provides a whole spectrum of options. We can take low-level control with `malloc` and `free`, create our own allocation functions, use raw `new` and `delete`, overload `new` and `delete` globally or on a per-type basis, pass extra arguments to `new` and `delete`, use "placement `new`" to allocate at a particular address, or even write "owning types" to avoid almost all of the manual allocation and deallocation code.

There's a ton of power, and a fair bit of complexity, here, but at no point must we give up any language features in order to move to higher-level or lower-level memory management strategies. We'll see some of those (very commonly-used) higher-level techniques later in the book when we cover the C++ Standard Library.

20. Implicit Type Conversion

When Implicit Type Conversion Happens

Both C# and C++ feature implicit type conversion, but there are many language-specific differences. In C++, implicit conversion occurs when using a wider variety of language features.

First up, and just like in C#, it happens when calling a function with a type other than the type of the function's parameter:

```
void Foo(float x)
{
}
Foo(1); // int -> float
```

Similarly, and also in C#, it happens when returning a value whose type isn't the function's return type:

```
float Bar()
{
    return 1; // int -> float
}
```

All the boolean logic operators require `bool` operands, so any non-`bool` needs conversion. This can be implicit in C++, but not C#:

```
bool b1 = !1; // int -> bool
bool b2 = false || 1; // int -> bool
```

```
bool b3 = true && 1; // int -> bool
```

The same goes for conditionals in C++, but not C#:

```
if (1) // int -> bool
{
}

bool b4 = 1 ? false : true; // int -> bool
```

And for C++ loops, not not C# loops:

```
while (1) // int -> bool
{
}
for (; 1; ) // int -> bool
{
}
do
{
} while (1); // int -> bool
```

The `switch` statement requires an integral type, so there's conversion required when using anything else. Both languages support this:

```
switch (false) // bool -> int
{
}
```

The C++ `delete` operator only deletes typed pointers. Here we have a user-defined conversion operator that converts a struct to an `int*`:

```
struct ConvertsToIntPointer
{
    operator int*() { return nullptr; }
};
delete ConvertsToIntPointer{}; // ConvertsToIntPointer ->
int*
```

Finally, both of C++'s `noexcept` and `explicit` can be conditional and require a `bool`:

```
void Baz() noexcept(1) // int -> bool
{
}

// C++20 and later
struct HasConditionalExplicit
{
    explicit(1) HasConditionalExplicit() {} // int ->
bool
};
```

Standard Conversions

We've already talked about user-defined conversions, but these aren't the only ways to implicitly convert between types. The language itself has many "standard" conversions that don't require us to write any code.

Usually these change the type itself, but in a few cases they just change its classification:

```
int x = 123;
const int y = x; // int -> const int
                 // also, lvalue -> rvalue
```

It's always OK to treat a non-`const` as a `const` since that just adds restriction. We can't go the other way as that would remove the `const` restriction.

Similarly, we can go from non-throwing functions to possibly-throwing but not the other way:

```
void DoStuff() noexcept
{
    throw 1;
}

void (*pFuncNoexcept)() noexcept = &DoStuff;
void (*pFunc)() = pFuncNoexcept; // non-throwing ->
possibly-throwing
```

```
void (*pFuncNoexcept2)() noexcept = pFunc; // Compiler
error
```

With those out of the way, all the rest of the standard conversions will change the type. First up we have function-to-pointer conversions. The previous example “took the address” of `DoStuff` to get a pointer to it, as we’ve seen before, but the is optional because there’s a standard conversion from functions to function pointers:

```
void DoStuff()
{
}

void (*pFunc)() = DoStuff; // function -> function
pointer
```

Note that this doesn’t work on non-`static` member functions as they require a class instance in order to implicitly pass the `this` argument:

```
struct Vector2
{
    float X;
    float Y;

    float SqrMagnitude() const noexcept
    {
        return X*X + Y*Y;
    }
```

```
};

Vector2 vec{2, 4};
// All of these are compiler errors:
float (*sqrMagnitude1)() = Vector2::SqrMagnitude
float (*sqrMagnitude2)() = vec.SqrMagnitude;
float (*sqrMagnitude3)(Vector2*) = Vector2::SqrMagnitude
float (*sqrMagnitude4)(Vector2*) = vec.SqrMagnitude;
```

Next we have array-to-pointer conversions:

```
int arr[]{1, 2, 3};
int* pArray = arr; // int[3] -> int*
```

This is known as “array to pointer decay” and it’s allowed because the two concepts are very similar. Semantically, it’s kind of like pointers are the “base class” of arrays. They’re both essentially a pointer and can be treated like an array (`x[123]`), but arrays are of a known size (`sizeof(arr) == sizeof(int)*3`). So one way to think about this is like an “upcast” from a derived class (array more information) to a base class (pointer with less information).

When it comes to numbers, we have two broad categories: promotion and conversion. Promotion won’t change the value of the number, but conversion might. Promotion generally increases the size of a number since larger sizes can represent all the values of smaller sizes. The same happens in C# when we, for example, pass a `short` to a function that takes an `int`.

This is commonly needed since all the arithmetic operators (e.g. `x + y`) require an `int` or larger. So the smaller primitive types will be

promoted to an `int` for all these operators as well as for all the reasons we saw above.

First, `signed char` is always promoted to `int`:

```
signed char c = 'A';
int i = c + 1; // c is promoted from 'signed char' to int
DebugLog(i); // 66 (ASCII for 'B')
```

The sizes of `char`, `unsigned char`, `unsigned short`, and `int` depend on factors such as the compiler and CPU architecture. If `int` can hold the full range of values for `char`, `unsigned char`, `unsigned short`, and `char8_t`, which is usually the case, they're promoted to `int`. If it can't, they're promoted to `unsigned int`.

```
unsigned char c = 'A'; // or 'unsigned short' or
'char8_t'
auto i = c + 1; // c is promoted from 'unsigned char' to
int or 'unsigned int'
DebugLog(i); // 66 (ASCII for 'B')
```

The compiler also determines the size of `wchar_t`. It, as well as `char16_t` and `char32_t`, will be promoted to the first type that's big enough to hold the full range of values:

1. `int`
2. `unsigned int`
3. `long`
4. `unsigned long`

5. `long long`
6. `unsigned long long`

```
wchar_t c = 'A';
auto i = c + 1; // c is promoted from 'wchar_t' to at
least an int
DebugLog(i); // 66 (ASCII for 'B')
```

Unscoped enumerations that don't have a fixed underlying type are also promoted into the same list of types:

```
enum Color // No underlying type
{
    Red,
    Green,
    Blue
};

Color c = Red;
auto i = c + 1; // c is promoted from 'Color' to at least
an int
DebugLog(i); // 1
```

If it does have a fixed underlying type, it's promoted to that type and then that type can be promoted:

```
enum Color : int // Has an underlying type
{
    Red,
    Green,
    Blue
};

Color c = Red;
long i = c + 1L; // c is promoted from 'Color' to int and
then to long
DebugLog(i); // 1
```

Bit fields will be promoted to the smallest size that can hold the full value range of the bit field, but it's a short list:

1. `int`
2. `unsigned int`

```
struct ByteBits
{
    bool Bit0 : 1;
    bool Bit1 : 1;
    bool Bit2 : 1;
    bool Bit3 : 1;
    bool Bit4 : 1;
    bool Bit5 : 1;
    bool Bit6 : 1;
    bool Bit7 : 1;
```

```
};

ByteBits bb{0};
int i = bb.Bit0 + 1; // bit field is promoted from 1 bit
to int
DebugLog(i); // 1
```

The `bool` type is promoted to `int` with `false` becoming `0` and `true` becoming `1` (not just non-zero). This isn't allowed in C#:

```
bool b = true;
int i = b + 1; // b is promoted from bool to int with
value 1
DebugLog(i); // 2
```

The `float` type is promoted to `double`, which works in both languages:

```
float f = 3.14f;
double d = f + 1.0; // f is promoted from float to double
DebugLog(d); // 4.14
```

Everything else is a numeric *conversion*, not *promotion*. Unlike promotion, the value may change during conversion.

First, there's conversion to an unsigned integer type. The result is smallest unsigned value modulus 2^n where `n` is the number of bits in the destination type. If the source type was signed, it's sign-extended

or truncated. If it was unsigned, it's zero-extended or truncated. This isn't allowed in C#:

```
int32_t si = 257;
uint8_t ui = si; // si is converted from int32_t to
uint8_t
                 // ui = 257 % 2^8 = 257 % 256 = 1
DebugLog(ui); // 1
```

When converting to a signed integer type, the value won't be changed if it can be represented in the destination type. Otherwise, the value was implementation-defined until C++20. Since C++20, the value is required by the C++ Standard to be computed like the conversion to unsigned: the source value modulus 2^n where `n` is the number of bits in the destination type. This also isn't allowed in C#:

```
uint32_t ui1 = 123;
int8_t si1 = ui1; // ui1 is converted from uint32_t to
int8_t
                  // The value doesn't change since it
can be held in int8_t
DebugLog(si1); // 1

uint32_t ui2 = 257;
int8_t si2 = ui2; // ui2 is converted from uint32_t to
int8_t
                  // Implementation-defined value until
C++20
                  // Since C++20:
```

```
                    // si2 = 257 % 2^8 = 257 % 256 = 1
DebugLog(si2); // 1 in C++20, unknown in C++17 and before
```

We saw above that when `bool` must become an `int`, it's promoted from `false` to `0` and `true` to `1`. For all other integer types, this is technically a conversion but it generates the same result. Despite not losing any precision like the above conversions, C# also forbids this:

```
bool b = true;
long i = b + 1; // b is converted from bool to long with
value 1
DebugLog(i); // 2
```

If a floating point type needs to become another floating point type, it's value is preserved exactly if that's possible in the destination type. If that's not possible and the source value is between two values in the destination type, one of them will be chosen. Usually the nearest value is chosen. Otherwise, the conversion is undefined behavior. This is also forbidden by C#:

```
double d = 3.14f;
float f = d; // d is converted from double to float
DebugLog(f); // 3.14
```

When converting a floating point type to an integer type, the fractional part is discarded. If the value can't fit, it's undefined behavior. Unlike above, there's no modulus applied for unsigned integer types. This too won't work in C#:

```
float f1 = 3.14f;
int8_t i1 = f1; // f1 is converted from float to int8_t
                // Fractional part (0.14) is discarded
DebugLog(i1); // 3

float f2 = 257.0f;
uint8_t i2 = f2; // f2 is converted from float to int8_t
                 // Value won't fit. Modulus not applied
                 // This is undefined behavior!
DebugLog(i2); // Could be anything!
```

The other way around, integers converted to floating point, works differently. They can be converted to any floating point type. The integer's value is preserved exactly if that's possible in the floating point type. Otherwise, if the integer's value is between two values in the floating point type then one of those two values will be chosen and that's usually the nearest value. Otherwise, the integer value won't fit and that's undefined behavior. C# allows this, too. For `bool`, we simply get `0` or `1` as with conversions to integer types, but not in C# as it's disallowed there.

```
int8_t i = 123;
float f1 = i; // i is converted from int8_t to float
DebugLog(f1); // 123

bool b = true;
float f2 = b; // b is converted from bool to float
DebugLog(f2); // 1
```

A “null pointer constant” in C++ is any integer literal with the value `0`, any constant with the value `0`, or `nullptr`. These can all be converted to any pointer type. Only `null` is allowed in C#.

```
int* p1 = 0; // Integer constant with value 0 is
converted to int*
DebugLog(p1); // 0

int* p2 = nullptr; // nullptr is converted to int*
DebugLog(p2); // 0
```

Similar to the “decaying” we saw with the array-to-pointer conversion, the shedding of `noexcept`, and the adding of `const`, all pointer types convert to `void*` since it’s a “pointer to anything.” This is allowed in both languages:

```
int x = 123;
int* pi = &x;
void* pv = pi; // int* is converted to void*
DebugLog(pv == pi); // true
```

As we’ve seen before when discussing inheritance, derived class pointers convert to base class pointers. The result points to the base class subobject of the derived class. Similarly, C# allows this with class references.

```
struct Vector2
{
    float X;
```

```
    float Y;
};

struct Vector3 : Vector2
{
    float Z;
};

Vector3 vec{};
vec.X = 1;
vec.Y = 2;
vec.Z = 3;
Vector3* pVec3 = &vec;
Vector2* pVec2 = pVec3; // Vector3* is converted to
Vector2*
DebugLog(pVec2->X, pVec2->Y); // 1, 2
```

Pointers to non-`static` members of base classes can likewise be converted to pointers to non-`static` members of derived classes:

```
float Vector2::* pVec2X = &Vector2::X;
float Vector3::* pVec3X = pVec2X; // Pointer to base
member is converted
                                  // to pointer to
derived member
Vector3 vec{};
vec.X = 1;
vec.Y = 2;
```

```
vec.Z = 3;
Vector3* pVec3 = &vec;
Vector2* pVec2 = pVec3;
DebugLog((*pVec2).*pVec2X, (*pVec3).*pVec3X); // 1, 1
```

Note that this isn't allowed for `virtual` inheritance:

```
struct Vector2
{
    float X;
    float Y;
};

struct Vector3 : virtual Vector2 // Virtual inheritance
{
    float Z;
};

float Vector2::* pVec2X = &Vector2::X;
float Vector3::* pVec3X = pVec2X; // Compiler error
```

Finally, all integer types, floating point types, unscoped enumerations, pointers, and pointers to members can be converted to `bool`. Zero and null become `false` and everything else becomes `true`. C# doesn't allow any of these.

```
int i = 123;
bool b1 = i; // int is converted to bool
```

```
DebugLog(b1); // true

float f = 3.14f;
bool b2 = f; // float is converted to bool
DebugLog(b2); // true

Color c = Red;
bool b3 = c; // Color is converted to bool
DebugLog(b3); // false

int* p = nullptr;
bool b4 = p; // int* is converted to bool
DebugLog(b4); // false

float Vector2::* pVec2X = &Vector2::X;
bool b5 = pVec2X; // Pointer to member is converted to
bool
DebugLog(b5); // true
```

Conversion Sequences

Now that we know about promotions and conversions, let's see how they're sequenced in order to change types. First, C++ has a "standard conversion sequence" that consists of the following steps which mostly don't apply to C#:

1) Zero or one conversions from an lvalue to an rvalue, array to pointer decays, and function to pointer conversions
2) Zero or one promotions or conversions of a number
3) Zero or one function pointer conversions, including non-throwing to possibly-throwing (only allowed in C++17 and later)
4) Zero or one non-`const` to `const` conversion

C++ also has an "implicit conversion sequence" with these steps:

1) Zero or one standard conversion sequences
2) Zero or one user-defined conversions
3) Zero or one standard conversion sequences

When we're passing an argument to a class' constructor or to a user-defined conversion function, we can only use the standard conversion sequence. That cuts out the possibility of calling a user-defined conversion operator:

```
struct MyClass
{
    MyClass(const int32_t)
    {
    }
};
```

```
uint8_t i1{123};
MyClass mc{i1}; // 1) lvalue to rvalue
                // 2) Promotion from uin8_t to uint32_t
                // 3) N/A
                // 4) uint8_t to 'const uint8_t'

struct C
{
};

struct B
{
    operator C()
    {
        return C{};
    }
};

struct A
{
    operator B()
    {
        return B{};
    }
};

// Compiler error: user-defined conversion operators not
```

```
allowed here
C c = A{};
```

Otherwise, we're allowed to use implicit conversion sequences. Here's a class that automatically closes files but converts to a `FILE*` so it can be used with a wide variety of functions in the C++ Standard Library, such as `fwrite` that writes to a file:

```
class File
{
    FILE* handle;

public:

    File(const char* path, const char* mode)
    {
        handle = fopen(path, mode);
    }

    ~File()
    {
        fclose(handle);
    }

    operator FILE*()
    {
        return handle;
    }
```

```
};

void Foo()
{
    File writer{"/path/to/file", "w"};
    char msg[] = "hello";

    // fwrite looks like this:
    //   std::size_t fwrite(
    //     const void* buffer,
    //     std::size_t size,
    //     std::size_t count,
    //     std::FILE* stream);

    // Last argument is implicitly converted from File to
FILE*
    fwrite(msg, sizeof(msg), 1, writer);

    // Note: File destructor called here to close the
file
}
```

Overflows

Integer math may result in an “overflow” where the result doesn’t fit into the integer type. C++ doesn’t have C#’s `checked` and `unchecked` contexts. Instead, it handles overflow differently depending on whether the math is signed or unsigned.

For signed math, an overflow is undefined behavior:

```
int32_t a = 0x7fffffff;
int32_t b = a + 1; // Overflow. Undefined behavior!
DebugLog(b); // Could be anything!
```

This isn’t as catastrophic as it may seem. The compiler will usually just generate an addition instruction and the overflow will be handled according to the CPU architecture’s rules for overflow. Only in cases where the compiler can prove a signed integer overflow will happen, like this example, is it likely to generate unexpected CPU instructions. It may also generate a compiler warning to bring this to the attention of the programmer. Usually the result ends up being the same in C# and C++, but technically it doesn’t have to.

Unsigned math is more forgiving. An overflow is simply performed modulus 2^n where `n` is the number of bits in the integer type:

```
uint8_t a = 255;
uint8_t b = a + 1; // Overflow. b = (255 + 1) % 256 = 0.
DebugLog(b); // 0
```

Arithmetic

We've seen a lot of promotion and conversion due to arithmetic already, but only covered simple cases so far. There are quite a few more rules for determining which operands are promoted or converted and what the "common type" arithmetic is performed on should be.

First of all, C++20 deprecates mixing floating point and enum types or enum types with other enum types. These were never allowed in C#.

```
enum Color
{
    Red,
    Green,
    Blue
};

// Deprecated: mixed enum and float
auto a = Red + 3.14f;

enum RangeType
{
    Melee,
    Distance
};

// Deprecated: mixed enum types
auto b = Red + Melee;
```

Integers get promoted first. Then, for all the binary operators except shifts, a book of specific type changes occur. First, if either operand is a `long double` then the other operand is converted to a `long double`. The same happens for `double` and `float`. C# has essentially the same behavior.

```
int i = 123;
long double ld = 3.14;
long double sum1 = ld + i; // i is converted from int to
'long double'
DebugLog(sum1); // 126.14

double d = 3.14;
double sum2 = d + i; // i is converted from int to double
DebugLog(sum2); // 126.14

float f = 3.14f;
double sum3 = f + i; // i is converted from int to float
DebugLog(sum3); // 126.14
```

For signed and unsigned integers, we need to consider the “conversion rank” of the types involved:

1. `bool`
2. `signed char`, `unsigned char`, and `char`
3. `short` and `unsigned short`
4. `int` and `unsigned int`
5. `long` and `unsigned long`

6. `long long` and `unsigned long long`

`char8_t`, `char16_t`, `char32_t`, and `wchar_t` have the same conversion rank as their underlying type, which depends on factors like the compiler, OS, and CPU architecture.

With that in mind, the operand with lower conversion rank is converted to the type of the operand with the greater conversion rank if both operands are either signed or unsigned. C# behaves the same way.

```
unsigned char uc = 'A'; // Coversion rank = 2
unsigned long ul = 1; // Conversion rank = 5

// uc has lower conversion rank, so it's converted to
unsigned long
unsigned long sum = uc + ul;
DebugLog(sum); // 66 (ASCII for 'B')
```

Otherwise, one operand is signed and the other is unsigned. C# doesn't allow this, but C++ does. In this case, if the unsigned operand's conversion rank is greater than or equal to the signed operand's conversion rank then the signed operand is converted to the type of the unsigned operand:

```
short s = 123; // Coversion rank = 3
unsigned long ul = 1; // Conversion rank = 5

// s has lower conversion rank, so it's converted to
unsigned long
```

```
unsigned long sum = s + ul;
DebugLog(sum); // 124
```

If that's not the case but the signed type can represent all the values of the unsigned type, the unsigned operand converted to the type of the signed operand:

```
long l = 123; // Coversion rank = 5
unsigned short us = 1; // Conversion rank = 3

// l has greater conversion rank and long can represent
all the
// values of 'unsigned short', so us is converted to long
long sum = l + us;
DebugLog(sum); // 124
```

And if that's not the case either, both operands are converted to the unsigned counterpart of the signed type:

```
long l = 123; // Coversion rank = 5
unsigned int ui = 1; // Conversion rank = 4

// Assume int and long are both 4 bytes (e.g. Windows)
// l has greater conversion rank but long can't represent
all the
// values of 'unsigned int', so l is converted from long
to unsigned long
// and ui is converted from unsigned int to unsigned long
```

```
unsigned long sum = l + ui;
DebugLog(sum); // 124
```

Narrowing Conversions

So far we've seen types either stay the size size or get bigger. Sometimes, the types get smaller or otherwise lose precision. These are called "narrowing" conversions and they're a subset of the implicit conversions we saw above. C# never allows these, but C++ does. For example, the conversion from floating point to integer loses precision by truncating the fractional part:

```
float f = 3.14f;
int i = f;
DebugLog(i); // 3
```

Converting from a larger floating point type to a smaller ones may also lose precision:

```
long double ld1 = 3.14;
double d1 = ld1; // 'long double' -> double
DebugLog(d1); // 3.14

long double ld2 = 3.14;
float f1 = ld2; // 'long double' -> float
DebugLog(f1); // 3.14

double d2 = 3.14;
float f2 = d2; // double -> float
DebugLog(d2); // 3.14
```

Converting from an integer or enumeration to a floating point type might also lose precision if the floating point type can't exactly represent the integer:

```
uint64_t i = 0xffffffffffffffff;
float f = i; // uint64_t -> float
DebugLog(f); // 1.84467e+19
uint64_t i2 = f;
DebugLog(i == i2); // false
```

These narrowing conversions are allowed by copy initialization as we've seen here, but forbidden by aggregate and list initialization:

```
// Compiler error: aggegate initialization with narrowing
int i1{3.14f};
IntHolder i2{3.14f};
int i3 = {3.14f};

// Compiler error: list initialization with narrowing
int i4[] = { 3.14f, 3.14f };

// OK: copy initialization with narrowing
int i5 = 3.14f; // copy
int i6(3.14f); // copy
```

Avoiding narrowing is yet-another reason to prefer initializing with curly braces.

Conclusion

This concludes the deep dive into C++'s implicit type conversion system. It's a lot more permissive than C#. That allows our code to be more terse, but also opens us up to a lot of potential bugs. So it's important that we understand all the rules from numeric promotion to conversion ranks to overflow handling.

21. Casting and RTTI

const_cast

C# has only one kind of cast: `(DestinationType)sourceType`. Since it's the only option, it must be capable of handling every possible reason for casting. C++ takes a different tack. It provides a suite of casts from which we may choose to suit the intention of particular casting operations.

The first cast in this suite is one of the simplest: `const_cast`. We use this when we simply want to treat a `const` pointer or references as non-`const`:

```
// Remove const from a pointer
int x = 123;
int const * constPtr = &x;
int* nonConstPtr = const_cast<int*>(constPtr);
*nonConstPtr = 456;
DebugLog(x); // 456

// Remove const from a reference
int const & constRef = x;
int& nonConstRef = const_cast<int&>(constRef);
nonConstRef = 789;
DebugLog(x); // 789

// It's OK to cast null
constPtr = nullptr;
```

```
int* nullPtr = const_cast<int*>(constPtr);
DebugLog(nullPtr); // null
```

Note that function pointers and member function pointers can't be `const_cast`.

In the first two examples, we used `const_cast` to get a non-`const` pointer and reference and then modified the value they referred to: `x`. That's perfectly OK because `x` wasn't actually `const`. However, it's undefined behavior if we modify a variable that's actually `const`:

```
int const x = 123;
int const & constRef = x;
int& nonConstRef = const_cast<int&>(constRef);
nonConstRef = 789; // Undefined behavior: modifying const
variable x
DebugLog(x); // Could be anything!
```

So what does `const_cast` actually do? It's really just a compile-time operation to reclassify an expression as non-`const`. No CPU instructions are generated because the CPU has no concept of `const`. In this sense, `const_cast` is "free" from a performance perspective.

reinterpret_cast

The next kind of cast is `reinterpret_cast`. This is another "free" cast that generates no CPU instructions. Instead, it just tells the compiler to "reinterpret" the type of one expression as another type.

We can only use this in particular situations. First, a pointer can be converted to and from an integer as long as that integer is large enough to hold all possible pointer values:

```
// Pointer -> Integer
int x = 123;
int* p = &x;
uint64_t i = reinterpret_cast<uint64_t>(p);
DebugLog(i); // memory address of x

// Integer -> Pointer
int* p2 = reinterpret_cast<int*>(i);
*p2 = 456;
DebugLog(x); // 456
```

We can also reinterpret `nullptr` as the integer `0`:

```
uint64_t i = reinterpret_cast<uint64_t>(nullptr);
DebugLog(i); // 0
```

More commonly, we can reinterpret one kind of pointer as another kind of pointer:

```
struct Vector2
{
    float X;
    float Y;
};

struct Point2
{
    float X;
    float Y;
};

Point2 point{2, 4};
Point2* pPoint = &point;
Vector2* pVector = reinterpret_cast<Vector2*>(pPoint);
DebugLog(pVector->X, pVector->Y); // 2, 4
```

We have to take some precautions in order to use the resulting pointer safely. First, CPUs have alignment requirements on various data types such as `float`. The destination type's alignment requirements can't be stricter than the source type's alignment requirements. It's up to us as programmers to know the alignment requirements for our target CPU architectures and ensure that we're using `reinterpret_cast` responsibly.

The C++ Standard says using the result of a `reinterpret_cast` is undefined behavior *except* in certain particular cases. The first is if they're "similar." That's defined as being the same type, pointers to the same type, pointers to members of the same class and those

members are similar, or arrays of the same size (or one has unknown size) with similar elements. Here are some examples:

```
// Similar: pointer to same type
int x = 123;
int* p = reinterpret_cast<int*>(&x); // int* -> int*
DebugLog(*p); // 123

// Similar: array with same dimensions and same type of elements
int a1[3]{1, 2, 3};
int (&a)[3] = reinterpret_cast<int(&)[3]>(a1);
DebugLog(a[0], a[1], a[2]); // 1, 2, 3

// Not similar: different types of pointers
float* pFloat = reinterpret_cast<float*>(p);
DebugLog(*pFloat); // Undefined behavior
```

Because undefined behavior may be implemented by a compiler however it chooses, many compilers are less strict than the C++ Standard requires. The last example of an `int*` to `float*` conversion in particular is commonly allowed by compilers as a kind of “type punning” similar to what we saw with unions.

If the types aren’t “similar” then we have two more chances to avoid undefined behavior. First, if one type is the signed or unsigned version of the same type:

```
// int* -> unsigned int*
int x = 123;
```

```
unsigned int* p = reinterpret_cast<unsigned int*>(&x);
DebugLog(*p); // 123
```

And second, if we're reinterpreting as a `char`, `unsigned char` or, in C++17 and later, `std::byte`. These are specifically allowed so we can look at the byte representation of objects, such as for serialization to disk or a network:

```
// Print the bytes of a Vector2
Vector2 vec{2, 4};
char* p = reinterpret_cast<char*>(&vec);
DebugLog(p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7]);
```

static_cast

Next up we have our first cast that can generate CPU instructions: `static_cast`. The compiler checks a book of conditions to decide what a `static_cast` should do. The first check is to see if there's an implicit conversion sequence from the source type to the destination type or if the destination type can be direct-initialized from the source type. If either is the case, the `static_cast` behaves like we wrote `DestType tempVariable(sourceType)`:

```
struct File
{
    FILE* handle;

    File(const char* path, const char* mode)
    {
        handle = fopen(path, mode);
    }

    ~File()
    {
        fclose(handle);
    }

    operator FILE*()
    {
        return handle;
    }
};
```

```
File reader{"/path/to/file", "r"};
FILE* handle = static_cast<FILE*>(reader); // Implicit
conversion
```

Next, it checks if we're downcasting from a pointer or reference to a base class to a pointer or reference to a (non-virtually) derived class:

```
struct Vector2
{
    float X;
    float Y;
};

struct Vector3 : Vector2
{
    float Z;
};

Vector3 vec;
vec.X = 2;
vec.Y = 4;
vec.Z = 6;
Vector2& refVec2 = vec; // Implicit conversion from
Vector3& to Vector2&
Vector3& refVec3 = reinterpret_cast<Vector3&>(refVec2);
// Downcast
DebugLog(refVec3.X, refVec3.Y, refVec3.Z); // 2, 4, 6
```

We can also `static_cast` to `void` to explicitly discard a value. This is sometimes used to silence an “unused variable” compiler warning:

```
Vector2 vec{2, 4};
static_cast<void>(vec); // Discard the result of the cast
```

If there’s a standard conversion from the *destination type* to the *source type*, `static_cast` will reverse it. It won’t reverse any lvalue-to-rvalue conversions, array-to-pointer decay, function-to-pointer conversions, function pointer conversions, or `bool` conversions though.

```
int i = 123;
float f = static_cast<int>(i); // Undo standard
conversion: int -> float
DebugLog(f); // 123
```

We can also explicitly perform some implicit conversions with `static_cast`: lvalue-to-rvalue, array-to-pointer decay, and function-to-pointer:

```
void SayHello()
{
    DebugLog("hello");
}

// lvalue to rvalue conversion
int i = 123;
```

```
int i2 = static_cast<int&&>(i);
DebugLog(i2); // 123

// Array to pointer decay
int a[3]{1, 2, 3};
int* p = static_cast<int*>(a);
DebugLog(p[0], p[1], p[2]); // 1, 2, 3

// Function to pointer conversion
void (*pFunc)() = static_cast<void(*)()>(SayHello);
pFunc(); // hello
```

Scoped enumerations can be converted to integer or floating point types using `static_cast`. Since C++20, this works like an implicit conversion from the enum's underlying type to the destination type. Before that, casting to `bool` was treated differently since only `0` would become `false` and everything else would become `true`.

```
enum class Color
{
    Red,
    Green,
    Blue
};

Color green{Color::Green};

// Scoped enum -> int
int i = static_cast<int>(green);
```

```
DebugLog(i); // 1

// Scoped enum -> float
float f = static_cast<float>(green);
DebugLog(f); // 1
```

We can go the other way, too: integers and floating point types can be `static_cast` to scoped or unscoped enumerations. We can also cast between enumeration types:

```
// Integer -> enum
int i = 1;
FgColor g1 = static_cast<FgColor>(i);
DebugLog(g1); // Green

// Floating point -> enum
float f = 1;
FgColor g2 = static_cast<FgColor>(f);
DebugLog(g2); // Green

// Cast between enum types
FgColor g3{FgColor::Green};
BgColor g4 = static_cast<BgColor>(g3);
DebugLog(g4); // Green
```

It's undefined behavior if the underlying type of the enum isn't fixed and the value being cast to the enum is out of its range. If it is fixed, the result is just like converting to the underlying type. Floating point values are first converted to the underlying type.

We can also use `static_cast` to upcast from a pointer to a member in a derived class to a pointer to a member in the base class:

```
float Vector3::* p1 = &Vector3::X;
float Vector2::* p2 = static_cast<float Vector2::*>(p1);
Vector3 vec;
vec.X = 2;
vec.Y = 4;
vec.Z = 6;
DebugLog(vec.*p1, vec.*p2); // 2, 2
```

And finally, `static_cast` can be used like `reinterpret_cast` to convert a `void*` to any other pointer type. The same caveats about alignment and type similarity apply.

```
int i = 123;
void* pv = &i;
int* pi = static_cast<int*>(pv);
DebugLog(*pi); // 123
```

C-Style Cast and Function-Style Cast

A “C-style” cast looks like a cast in C as well as C#: `(DestinationType)sourceType`. It behaves quite differently in C++ compared to C#. In C++, it’s mostly a shorthand for the first “named” cast whose prerequisites are met in this order:

1. `const_cast<DestinationType>(sourceType)`
2. `static_cast<DestinationType>(sourceType)` with more leniency: pointers and references to or from derived classes or members of derived classes can be cast to pointers or references to base classes or members of base classes
3. `static_cast` (with more leniency) then `const_cast`
4. `reinterpret_cast<DestinationType>(sourceType)`
5. `reinterpret_cast` then `const_cast`

```
// Uses const_cast (#1)
int const i1 = 123;
int i2 = (int)i1;
DebugLog(i2); // 123

// Uses static_cast (#2)
Vector2 vec{2, 4};
Vector3* pVec = (Vector3*)&vec;
DebugLog(pVec->X, pVec->Y); // 2, 4 (undefined behavior
to use Z!)

// Uses static_cast the const_cast (#3)
Vector2 const * pConstVec = &vec;
```

```
Vector3* pVec3 = (Vector3*)pConstVec;
DebugLog(pVec3->X, pVec3->Y); // 2, 4 (undefined behavior
to use Z!)

// Uses reinterpret_cast (#4)
float* f1 = (float*)&i2;
DebugLog(*f1); // 1.7236e-43

// Uses reinterpret_cast then const_cast (#5)
float* f2 = (float*)&i1;
DebugLog(*f2); // 1.7236e-43
```

A “function-style” cast works just like a C-style cast. It looks like a function call and even requires the type to have only one word: `int` not `unsigned int`. Be careful not to mistake it for a function call or class initialization.

```
int i = 123;
float f = float(i);
DebugLog(f); // 123
```

dynamic_cast

All of the casts we've seen so far are "static." That means the way they operate is determined at compile time and don't depend on the run-time value of the expression being cast. For example, consider this downcast:

```
void PrintZ(Vector2& vec)
{
    // Downcast
    Vector3& refVec3 = reinterpret_cast<Vector3&>(vec);

    // Undefined behavior if vec isn't really a Vector3
    DebugLog(refVec3.X, refVec3.Y, refVec3.Z);
}
```

Remember that `reinterpret_cast` generates no CPU instructions. This means the compiler isn't generating any CPU instructions that would check if `vec` is really a `Vector3`. If it is, this code works just fine. If it's not, reading `Z` will read the four bytes that come after wherever the `Vector2` is in memory. That's almost certainly not a valid `Z` value and will cause severe errors in our program logic when we try to use it that way. It's also undefined behavior, so the compiler might generate surprising CPU instructions such as just skipping reading and printing `Z` altogether.

To address this issue, C++ has a "safe" cast called `dynamic_cast`. It works very similarly to C#'s only cast.

Like `static_cast`, a sequence of checks is performed to decide what the CPU should do. First, we can cast to the same type or to add

`const`:

```
// Cast to same type
Vector2 v{2, 4};
Vector2& r1 = v;
Vector2& r2 = dynamic_cast<Vector2&>(r1);
DebugLog(r2.X, r2.Y); // 2, 4

// Cast to add const
Vector2 const & r3 = dynamic_cast<Vector2 const &>(r1);
DebugLog(r3.X, r3.Y); // 2, 4
```

Second, if the value is null then the result is null:

```
Vector2* p1 = nullptr;
Vector2* p2 = dynamic_cast<Vector2*>(p1);
DebugLog(p2); // 0
```

Third, we can upcast from a pointer or reference to a derived class to a pointer or reference to a base class:

```
Vector3 vec;
vec.X = 2;
vec.Y = 4;
vec.Z = 6;
Vector3& r3 = vec;
```

```
Vector2& r2 = dynamic_cast<Vector2&>(r3);
DebugLog(r2.X, r2.Y); // 2, 4
```

Fourth, we can cast pointers to classes that have at least one `virtual` function to `void*` and we'll get a pointer to the most-derived object that pointer points to:

```
struct Combatant
{
    virtual ~Combatant()
    {
    }
};

struct Player : Combatant
{
    int32_t Id;
};

Player player;
player.Id = 123;
Combatant* p = &player;
void* pv = dynamic_cast<void*>(p); // Downcast to most-
derived class: Player*
Player* p2 = reinterpret_cast<Player*>(pv);
DebugLog(p2->Id); // 123
```

Finally, we have the primary use case of `dynamic_cast`: a downcast from a pointer or reference to a base class to a pointer or reference to a derived class. This generates CPU instructions that examine the object being pointed to or referenced by the expression to cast. If that object is really a base class of the destination type and that destination type has only one sub-object of the base class, which may not be the case with non-`virtual` inheritance, then the cast succeeds with a pointer or reference to the derived class:

```
Player player;
player.Id = 123;
Combatant* p = &player;
Player* p2 = dynamic_cast<Player*>(p); // Downcast
DebugLog(p2->Id); // 123
```

This can also be used to perform a “sidecast” from one base class to another base class:

```
struct RangedWeapon
{
    float Range;

    virtual ~RangedWeapon()
    {
    }
};

struct MagicWeapon
{
```

```
    enum { FireType, WaterType, ArcaneType } Type;
};

struct Staff : RangedWeapon, MagicWeapon
{
    const char* Name;
};

Staff staff;
staff.Name = "Staff of Freezing";
staff.Range = 10.0f;
staff.Type = MagicWeapon::WaterType;

Staff& staffRef = staff;
RangedWeapon& rangedRef = staffRef; // Implicit
conversion upcasts
MagicWeapon& magicRef = dynamic_cast<MagicWeapon&>
(rangedRef); // Sidecast
DebugLog(magicRef.Type); // 1
```

If neither the downcast nor the sidecast succeed, the cast fails. When pointers are being cast, the cast evaluates to a null pointer of the destination type. If references are being cast, a `std::bad_cast` exception is thrown:

```
struct Combatant
{
    virtual ~Combatant()
```

```
    {
    }
};

struct Player : Combatant
{
    int32_t Id;
};

struct Enemy : Combatant
{
    int32_t Id;
};

// Make a Combatant: the base class
Combatant combatant;
Combatant* pc = &combatant;
Combatant& rc = combatant;

// Cast fails. Combatant object isn't a Player. Null
returned.
Player* pp = dynamic_cast<Player*>(pc);
DebugLog(pp); // 0

try
{
    // Cast fails. Combatant object isn't a Player.
std::bad_cast thrown.
```

```
    Player& rp = dynamic_cast<Player&>(rc);
    DebugLog(rp.Id); // Never called
}
catch (std::bad_cast const &)
{
    DebugLog("cast failed"); // Gets printed
}
```

Note that using `dynamic_cast` on `this` during a constructor is undefined behavior unless the destination type is the same class type or a base class type. We'll see why in the next section.

Run-Time Type Information

In order to implement `dynamic_cast`, the compiler must generate what's known as Run-Time Type Information or RTTI. The exact format of this information is compiler-specific, but the compiler will generate data to be used at runtime by `dynamic_cast` in order to determine the type of a particular object.

Since `dynamic_cast` only works on types with at least one `virtual` function, it can take advantage of the object's `virtual` function table or "vtable." This is a compiler-generated array of function pointers for all the `virtual` functions of a class. One table will be generated for each class in the inheritance hierarchy. A pointer to the table, known as a "virtual table pointer" or "vpointer," will be added as a data member of all classes in the hierarchy and initialized during construction.

This virtual table pointer can therefore also be used to identify the class of an object since there is one virtual function table per class. The inheritance hierarchy is then conceptually expressed as a tree of virtual table pointers with implementation details varying by compiler.

Because all this RTTI data adds to the executable size, many compilers allow it to be disabled. That also disables `dynamic_cast` as it depends on RTTI.

typeid

There is one other use of RTTI: the `typeid` operator. It's used to get information about a type, similar to `typeof` or `GetType` in C#. The operand can be either be named statically like `typeof` in C# or dynamically like `GetType` in C# to look up the type based on an object's value. The C++ Standard Library's `<typeinfo>` header is required to use this.

```
// Static usage based on type
std::type_info const & ti1{typeid(Combatant)};

// Dynamic usage based on variable with a virtual
function
Enemy enemy;
std::type_info const & ti2{typeid(enemy)};

// Dynamic usage based on variable with no virtual
function
// Equivalent to static usage: typeid(int)
int i = 123;
std::type_info const & ti3{typeid(i)};
```

It evaluates to a `const std::type_info` which has only a few useful members:

- `operator==(const std::type_info&)` and `operator!=(const std::type_info&)` to compare types

- `std::size_t hash_code()` to get an integer that's always the same for a given type
- `const char* name()` to get the type's string name

When using `typeid` on a null pointer, a `bad_typeid` is thrown:

```
Enemy* pe = nullptr;
try
{
    // Doesn't dereference null
    // Instead, attempts to get the type_info for what pe
points to
    std::type_info const & ti{typeid(*pe)};

    // Not printed
    DebugLog(ti.name());
}
catch (std::bad_typeid const &)
{
    DebugLog("bad typeid call"); // Is printed
}
```

One common surprise with `typeid` is that it ignores `const`:

```
DebugLog(typeid(int) == typeid(const int)); // true
```

Another is that the `name` member function doesn't return any specific string. That string is also usually some compiler-specific code that

may or may not have the name of the type from the source code:

```
// All of these will vary from compiler to compiler
DebugLog(typeid(int).name()); // i
DebugLog(typeid(long).name()); // l
DebugLog(typeid(Enemy).name()); // 5Enemy
```

One more is that the `std::type_info` for one call might not be the same object as the `std::type_info` for another call, even if they're the same type. The `hash_code` member function should be used instead:

```
DebugLog(&typeid(int) == &typeid(int)); // Maybe false
DebugLog(typeid(int).hash_code() ==
typeid(int).hash_code()); // Always true
```

Conclusion

As is often the case when comparing the two languages, C++ provides many options when C# provides only a few. In the case of casting, C++ provides a wide variety of named, C-style, and function-style casts for specific purposes while C# essentially only provides `dynamic_cast`.

When used appropriately, this can make many casts “free” as no CPU instructions will be generated and no size will be added to the executable. When used inappropriately, undefined behavior may cause severe errors such as crashes and data corruption. It’s up to us as programmers to know the rules of casting and to judiciously choose the appropriate cast for our task. The consequences, even with thrown exceptions in C#, of careless casting really demand that we exercise caution regardless of language and cast type.

22. Lambdas

Basic Syntax

Syntactically, lambdas look different in C++ than they do in C#. First, there's no equivalent to C#'s "expression lambdas:" `(arg1, arg2, ...) => expr`. C++ only has the equivalent of C#'s "statement lambdas:" `(arg1, arg2, ...) => { stmnt1; stmnt2; ... }`. In their simplest form, they look like this:

```
[]{ DebugLog("hi"); }
```

The first part (`[]`) is the list of captures, which we'll go into deeply in a bit. The second part (`{ ... }`) is the list of statements to execute when the lambda is invoked.

Now let's add a parameters list:

```
[](int x, int y){ return x + y; }
```

Besides the capture list (`[]`) and the omission of an `=>` after the parameters list, this now looks just like a C# lambda. In the first form that omitted the parameters list, the lambda simply takes no parameters.

Note that, unlike all the named functions we've seen so far, there's no return type stated here. The return type is implicitly deduced by the compiler by looking at the type of our `return` statements. That's just like we've seen before when declaring functions with an `auto` return type or what we get in C#.

If we'd rather explicitly state the return type, we can do so with the "trailing" return type syntax:

```
[](int x, int y) -> int { return x + y; }
```

Like normal functions, we can also take `auto`-typed parameters:

```
[](auto x, auto y) -> auto { return x + y; }
[](auto x, auto y) { return x + y; } // Trailing return
type is optional
[](auto x, int y) { return x + y; } // Not every
parameter has to be auto
```

Lambda Types

So what type does a lambda expression have? In C#, we get a type that can be converted to a delegate type like `Action` or `Func<int, int, int>`. In C++, the compiler generates an unnamed class. It looks like this:

```
// Compiler-generated class for this lambda:
//   [](int x, int y) { return x + y; }
// Not actually named LambdaClass
class LambdaClass
{
    // Lambda body
    // Not actually named LambdaFunction
    static int LambdaFunction(int x, int y)
    {
        return x + y;
    }

public:

    // Default constructor
    // Only if no captures
    LambdaClass() = default;

    // Copy constructor
    LambdaClass(const LambdaClass&) = default;
```

```
    // Move constructor
    LambdaClass(LambdaClass&&) = default;

    // Destructor
    ~LambdaClass() = default;

    // Function call operator
    int operator()(int x, int y) const
    {
        return LambdaFunction(x, y);
    }

    // User-defined conversion function to function
pointer
    // Only if no captures
    operator decltype(&LambdaFunction)() const noexcept
    {
        return LambdaFunction;
    }
};
```

Since it's just a normal class, we can use it like a normal class. The only difference is that we don't know its name, so we have to use `auto` for its type:

```
void Foo()
{
    // Instantiate the lambda class. Equivalent to:
```

```
    //   LambdaClass lc;
    auto lc = [](int x, int y){ return x + y; };

    // Invoke the overloaded function call operator
    DebugLog(lc(200, 300)); // 500

    // Invoke the user-defined conversion operator to get
a function pointer
    int (*p)(int, int) = lc;
    DebugLog(p(20, 30)); // 50

    // Call the copy constructor
    auto lc2{lc};
    DebugLog(lc2(2, 3)); // 5

    // Destructor of lc and lc2 called here
}
```

Default Captures

So far, our lambdas have always had an empty list of captures: `[]`. In C#, captures are always implicit. In C++, we have much more control over what we capture and how we capture it.

To start, let's look at the most C#-like kind of capture: `[&]`. This is a "capture default" that says to the compiler "capture everything the lambda uses as a reference." Here's how it looks:

```
// Something outside the lambda
int x = 123;

// Default capture mode set to "by reference"
auto addX = [&](int val)
{
    // Lambda references "x" that's outside the lambda
    // Compiler captures "x" by reference: int&
    return x + val;
};

DebugLog(addX(1)); // 124
```

We can see that `x` is captured by reference by modifying `x` after we capture it:

```
int x = 123;

// Capture reference to x, not a copy of x
```

```
auto addX = [&](int val) { return x + val; };

// Modify x after the capture
x = 0;

// Invoke the lambda
// Lambda uses the reference to x, which is 0
DebugLog(addX(1)); // 1
```

If we don't like this behavior, we can switch the "capture default" to `[=]` which means "capture everything the lambda uses as a copy." Here's how that looks:

```
int x = 123;

// Capture a copy of x, not a reference to x
auto addX = [=](int val) { return x + val; };

// Modify x after the capture
// Does not modify the lambda's copy
x = 0;

// Invoke the lambda
// Lambda uses the copy of x, which is 123
DebugLog(addX(1)); // 124
```

While it's deprecated starting with C++20, it's important to note that `[=]` can implicitly capture a reference to the current object: `*this`.

Here's one way that happens:

```
struct CaptureThis
{
    int Val = 123;

    auto GetLambda()
    {
        // Default capture mode is "copy"
        // Lambda uses "this" which is outside the lambda
        // "this" is copied to a CaptureThis*
        return [=]{ DebugLog(this->Val); };
    }
};

auto GetCaptureThisLambda()
{
    // Instantiate the class on the stack
    CaptureThis ct{};

    // Get a lambda that's captured a pointer to "ct"
    auto lambda = ct.GetLambda();

    // Return the lambda. Calls the destructor for "ct".
    return lambda;
}

void Foo()
```

```
{
    // Get a lambda that's captured a pointer to "ct"
which has had its
    // destructor called and been popped off the stack
    auto lambda = GetCaptureThisLambda();

    // Dereference that captured pointer to "ct"
    lambda(); // Undefined behavior: could do anything!
}
```

This example happened to create a “dangling” pointer to `this`, but the same can happen with any other pointer or reference. It’s important to make sure that captured pointers and references don’t end their lifespan before the lambda does!

Individual Captures

The next kind of element we can add to a capture list is called an “individual capture” since it captures something specific from outside the lambda.

There are a few forms of individual capture. First up, we can simply put a name:

```
int x = 123;

// Individually capture "x" by copy
auto addX = [x](int val)
{
    // Use the copy of "x"
    return x + val;
};

// Modify "x" after the capture
x = 0;

DebugLog(addX(1)); // 124
```

If we want to initialize the captured copy, we can add any of the usual forms of initialization:

```
int x = 123;

// Individually capture "x" by copying it to a variable
```

```
named "a"
auto addX = [a = x](int val)
{
    // Use the copy of "x" via the "a" variable
    return a + val;
};

// Modify "x" after the capture
x = 0;

DebugLog(addX(1)); // 124
```

The captured variable can even have the same name as what it captures, similar to when we used just `[x]`:

```
[x = x](int val){ return x + val; };
```

Other initialization forms are also available. Here are a couple:

```
[a{x}](int val){ return a + val; };
[a(x)](int val){ return a + val; };
```

In contrast, we can individually capture by reference:

```
int x = 123;

// Individually capture "x" by reference
```

```
auto addX = [&x](int val)
{
    // Use the reference to "x"
    return x + val;
};

// Modify "x" after the capture
x = 0;

DebugLog(addX(1)); // 1
```

We can initialize individually-captured references, too:

```
int x = 123;

// Individually capture "x" by reference as a reference named "a"
auto addX = [&a = x](int val)
{
    // Use the reference to "x" via "a"
    return a + val;
};

// Modify "x" after the capture
x = 0;

DebugLog(addX(1)); // 1
```

Regardless of whether we capture by reference or by copy, we can initialize using arbitrary expressions rather than simply the name of a variable:

```
auto lambda = [a = 2+2]{ DebugLog(a); };
lambda(); // 4
```

We also have two ways to individually capture `this`. The first is just `[this]` which captures `this` by reference:

```
struct CaptureThis
{
    int Val = 123;

    int Foo()
    {
        // Capture "this" by reference
        auto lambda = [this]
        {
            // Use captured "this" reference
            return this->Val;
        };

        // Modify "Val" after the capture
        this->Val = 0;

        // Invoke the lambda
        // Uses reference to "this" which has a modified
```

```
Val
        return lambda();
    }
};

CaptureThis ct{};
DebugLog(ct.Foo()); // 0
```

The second way to capture `this` is with `[*this]`, which makes a copy of the class object:

```
struct CaptureThis
{
    int Val = 123;

    int Foo()
    {
        // Capture "this" by copy
        auto lambda = [*this]
        {
            // Use captured "this" copy
            return this->Val;
        };

        // Modify "Val" after the capture
        this->Val = 0;

        // Invoke the lambda
```

```
        // Uses copy of "*this" which has the original
Val
        return lambda();
    }
};

CaptureThis ct{};
DebugLog(ct.Foo()); // 123
```

Captured Data Members

So what does it mean when a lambda “captures” something? Mostly, it just means that data members are added to the lambda’s class and initialized via its constructor. Say we have this lambda:

```
[&m{multiply}, a{add}](float val){ return m*val + a; }
```

We can use the lambda like this:

```
float multiplyAndAddLoopLambda(float multiply, float add,
int n)
{
    // Capture "multiply" by reference as "m"
    // Capture "add" by copy as "a"
    auto madd = [&m{multiply}, a{add}](float val){ return
m*val + a; };

    float cur = 0;
    for (int i = 0; i < n; ++i)
    {
        cur = madd(cur);
    }
    return cur;
}

DebugLog(multiplyAndAddLoopLambda(2.0f, 1.0f, 5)); // 31
```

The reason is that the compiler generates a class for the lambda that looks like this:

```
// Compiler-generated class for this lambda:
//   [&m{multiply}, a{add}](float val){ return m*val + a;
}
// Not actually named LambdaClass
class LambdaClass
{
    // "Captures" of the lambda
    // Order is unspecified
    // Not actually named "m" and "a"
    float& m;
    const float a;

public:

    // Constructor
    // Initializes captures
    LambdaClass(float& multiply, float add)
        : m{multiply}, a{add}
    {
    }

    // Copy constructor
    LambdaClass(const LambdaClass&) = default;

    // Move constructor
```

```
    LambdaClass(LambdaClass&&) = default;

    // Destructor
    ~LambdaClass() = default;

    // Function call operator
    float operator()(float val) const
    {
        // Lambda body
        return m*val + a;
    }
};
```

Notice that the default constructor has been replaced by a constructor that initializes the captures, be they by reference or copy. If there's no capture initializer (`[x]` or `[&x]`), captures are direct-initialized. Otherwise, they're copy-initialized or direct-initialized as specified by the capture initializer (`[x{y}]` or `[x = y]`). Array elements are direct-initialized in sequential order.

Another change in this compiler-generated lambda class is that the user-defined conversion operator to a function pointer has been removed. That's because a plain function pointer doesn't have access to the `this` pointer required to get the captures it needs to do its work. It's as though we tried to write this:

```
float LambdaFunction(float val)
{
    // Compiler error: no "m"
    // Compiler error: no "a"
```

```
    return m*val + a;
}
```

Since we may need control over the modifiers placed on the lambda class' data members, we can add keywords like `mutable` and `noexcept` to the lambda and they'll be added to the data members too:

```
int x = 1;

// Compiler error
// LambdaClass::operator() is const and LambdaClass::x
isn't mutable
auto lambda1 = [x](){ x = 2; };

// OK: LambdaClass::x is mutable
auto lambda2 = [x]() mutable { x = 2; };
```

When we used the lambda above, the compiler generated code to use the lambda's class that looks more or less like this:

```
float multiplyAndAddLoopClass(float multiply, float add,
int n)
{
    // "Capture" the "multiply" and "add" variables as
data members of "madd"
    LambdaClass madd{multiply, add};
```

```
    float cur = 0;
    for (int i = 0; i < n; ++i)
    {
        cur = madd(cur);
    }
    return cur;
}

DebugLog(multiplyAndAddLoopClass(2.0f, 1.0f, 5)); // 31
```

Capture Rules

There are a number of language rules about how we can use captures. First, if the default capture mode is by reference, individual captures can't also be by reference:

```
int x = 123;

// Compiler error: can't individually capture by
reference when the default
//                 capture mode is by reference
auto lambda = [&, &x]{ DebugLog(x); };
```

Second, if the default capture mode is by copy then all individual captures must be by reference, `this`, or `*this`:

```
// Compiler error: can't individually capture by copy
when the default
//                 capture mode is by copy
auto lambda1 = [=, =x]{ DebugLog(x); };

auto lambda2 = [=, &x]{ DebugLog(x); }; // OK
auto lambda3 = [=, this]{ DebugLog(this->Val); }; // OK
auto lambda4 = [=, *this]{ DebugLog(this->Val); }; // OK
```

Third, we can only capture a single name or `this` once:

```
int x = 123;

// Compiler error: can't capture by name twice
auto lambda1 = [x, x]{ DebugLog(x); };

// Compiler error: can't capture by name twice (with
initialization)
auto lambda2 = [x, x=x]{ DebugLog(x); };

// Compiler error: can't capture by name twice (mixed
capture modes)
auto lambda3 = [x, &x]{ DebugLog(x); };

// Compiler error: can't capture "this" twice
auto lambda4 = [this, this]{ DebugLog(this->Val); };

// Compiler error: can't capture "this" twice (mixed
capture modes)
auto lambda5 = [this, *this]{ DebugLog(this->Val); };
```

Fourth, if the lambda isn't in a block or a class' default data member initializer, it can't use default captures or have individual captures without an initializer:

```
// Global scope...

// Compiler error: can't use default captures here
auto lambda1 = [=]{ DebugLog("hi"); };
```

```
auto lambda2 = [&]{ DebugLog("hi"); };

// Compiler error: can't use uninitialized captures here
auto lambda3 = [x]{ DebugLog(x); };
auto lambda4 = [&x]{ DebugLog(x); };
```

Fifth, class members can only be captured individually using an initializer:

```
class Test
{
    int Val = 123;

    void Foo()
    {
        // Compiler error: member must be captured with
an initializer
        auto lambda1 = [Val]{ DebugLog(Val); };

        auto lambda2 = [Val=Val]{ DebugLog(Val); }; // OK
        auto lambda3 = [&Val=Val]{ DebugLog(Val); }; //
OK
    }
};
```

Sixth, and similarly, class members are never captured by default capture modes. Only `this` is captured and members are accessed from that pointer.

```
class Test
{
    int Val = 123;

    void Foo()
    {
        // Member not captured by default capture mode
        // Only "this" is captured
        auto lambda1 = [=]{ DebugLog(Val); };
        auto lambda2 = [&]{ DebugLog(Val); };
    }
};
```

Seventh, lambdas in default arguments can't capture anything:

```
// Compiler error: lambda in default argument can't have
a capture
void Foo(int val = ([=]{ return 2 + 2; })())
{
    DebugLog(val);
}
```

Eigth, anonymous union members can't be captured:

```
union
{
    int32_t intVal;
```

```
    float floatVal;
};
intVal = 123;

// Compiler error: can't capture an anonymous union
member
auto lambda = [intVal]{ DebugLog(intVal); };
```

Ninth, and finally, if a nested lambda captures something that's captured by the lambda it's nested in, the nested capture is transformed in two cases. The first case is if the lambda it's nested in captured something by copy. In this case, the nested lambda captures the data member of outer lambda's class instead of what was originally-captured.

```
void Foo()
{
    int x = 1;
    auto outerLambda = [x]() mutable
    {
        DebugLog("outer", x);
        x = 2;
        auto innerLambda = [x]
        {
            DebugLog("inner", x);
        };
        innerLambda();
    };
    x = 3;
```

```
    outerLambda(); // outer 1 inner 2
}
```

The second case is if the lambda it's nested in captured something by reference. In this case, the nested lambda captures the original variable or `this`:

```
void Foo()
{
    int x = 1;
    auto outerLambda = [&x]() mutable
    {
        DebugLog("outer", x);
        x = 2;
        auto innerLambda = [&x]
        {
            DebugLog("inner", x);
        };
        innerLambda();
    };
    x = 3;
    outerLambda(); // outer 3 inner 2
}
```

IILE

A common idiom in C++, seen above in the default function argument example, is known as an Immediately-Invoked Lambda Expression. We can use these in a variety of situations to work around various language rules. For example, many C++ programmers strive to keep everything `const` that can be `const`. If, however, the value to initialize a `const` variable to requires multiple statements then it may be necessary to remove `const`. For example:

```
Command command;
switch (byteVal)
{
    case 0:
        command = Command::Clear;
        break;
    case 1:
        command = Command::Restart;
        break;
    case 2:
        command = Command::Enable;
        break;
    default:
        DebugLog("Unknown command: ", byteVal);
        command = Command::NoOp;
}
```

Here we couldn't make `command` into a `const` variable even though we may only be initializing it in the `switch` and never setting it

afterward. We could have transformed the `switch` into a chain of conditional operators, but then we wouldn't be able to print the error message in the `default` case:

```
const Command command = byteVal == 0 ?
    Command::Clear :
    byteVal == Command::Restart ?
        Command::Enable :
        Command::NoOp;
}

// Unnecessary branch instruction: we already determined
it's NoOp above
if (command == Command::NoOp)
{
    DebugLog("Unknown command: ", byteVal);
}
```

To get around this, we can use an IILE to wrap the `switch`. To do so, we put parentheses around the lambda and then parentheses afterward to immediately invoke it:

```
const Command command = ([byteVal]{
    switch (byteVal)
    {
        case 0: return Command::Clear;
        case 1: return Command::Restart;
        case 2: return Command::Enable;
```

```
        default:
            DebugLog("Unknown command: ", byteVal);
            return Command::NoOp;
    }})();
```

The compiler will then create an instance of the lambda class that's destroyed at the end of the statement. The overhead of the constructor and destructor will be optimized away, effectively making the IILE and the `const` it enables "free."

C# Equivalency

We've compared C++ lambdas to C# lambdas a little so far, but let's take a closer look. First, we've seen that only "statement lambdas" are supported in C++. We can't write a C# "expression lambda" like this:

```
// C#
(int x, int y) => x + y
```

This example also shows another difference: C# lambda parameters are always explicitly typed. C++ lambda parameters may be `auto` to support a variety of argument types:

```
auto lambda = [](auto x, auto y){ return x + y; };

// int arguments
DebugLog(lambda(2, 3)); // 5

// float arguments
DebugLog(lambda(3.14f, 2.0f)); // 5.14
```

Similarly, C++ return types may be `auto` and that is in fact the default when a trailing return type like `-> float` isn't used. C# lambdas must always have an implicit return type. To force it, a cast is typically used within the body of the lambda:

```
// C#
(float x, float y) => { return (int)(x + y); };
```

On the other hand, C# is more explicit than C++ when storing the lambda in a variable as `var` cannot be used:

```
// C#
Func<int, int, int> f1 = (int x, int y) => { return x + y;}; // OK
var f2 = (int x, int y) => { return x + y;}; // Compiler error
```

C++ allows for `auto`:

```
auto lambda = [](int x, int y){ return x + y; };
```

C# lambdas support discarding arguments:

```
// C#
Func<int, int, int> f = (int x, int _) => { return x; };
// Discard y
```

C++ can do that by either omitting the name, similar to `_`, or by casting the argument to `void`:

```
// Omit argument name
auto lambda1 = [](int x, int){ return x; };

// Cast argument to void
auto lambda2 = [](int x, int y){ static_cast<void>(y);
return x; };
```

C# has `static` lambdas to prevent capturing local variables or non-`static` fields. That's the default in C++. Capturing in C++ is opt-in via default and individual captures:

```
int x = 123;

// Capture nothing
// Compiler error: can't access x
auto lambda1 = []{ DebugLog(x); };

// Capture implicitly by copy
auto lambda2 = [=]{ DebugLog(x); };

// Capture implicitly by reference
auto lambda3 = [&]{ DebugLog(x); };

// Capture explicitly by copy
auto lambda4 = [x]{ DebugLog(x); };

// Capture explicitly by reference
auto lambda5 = [&x]{ DebugLog(x); };
```

C# forbids capturing `in`, `ref`, and `out` variables. C++ references and pointers, the closest match to C#, can be freely captured in a variety of ways.

C# supports `async` lambdas as it does with other kinds of functions. C++ has no built-in `async` and `await` system, so these are not supported.

Finally, and most significantly, C++ lambdas are not a delegate as they are in C#. C++ has no concept of managed types or any built-in construct that operates like a delegate with its garbage-collection and support for multiple listeners determined at runtime.

Instead, C++ lambdas are just regular C++ classes. They have constructors, assignment operators, destructors, overloaded operators, and user-defined conversion operators. As such, they behave like other C++ class objects rather than as managed, garbage-collected C# classes.

Conclusion

Lambdas in both languages fulfill a similar role: to provide unnamed functions. Aside from `async` lambdas in C#, the C++ version of lambdas offers a much broader feature set. The two languages' approaches diverge as C# makes the trade-off in favor of safety by making lambdas be managed delegates. C++ takes the low, or often zero, overhead approach of using regular classes at the cost of possible bugs such as dangling pointers and references.

23. Compile-Time Programming

Constant Variables

C# has `const` fields of classes and structs. They must be immediately initialized by a constant expression, which is one that's evaluated at compile time. Their type must be a primitive like `int` and `float` or `string`. A `const` is implicitly `static` and `readonly`.

Likewise, C++ has `constexpr` variables. They must be "literal types" which include primitives like `int` and `float`, but also references, classes that meet certain criteria, arrays of "literal types," or `void`. They are implicitly `const`, but not `static`.

```
struct MathConstants
{
    // Constant member variable
    // Implicitly `const`
    // Not implicitly `static`. Need to add the keyword.
    static constexpr float PI = 3.14f;
};

// Constant global variable
constexpr int32_t numGamesPlayed = 0;

void Foo()
{
    // Constant local variable
    constexpr int32_t expansionMultiplier = 3;
```

```
    // Constant reference
    // Need to add `const` because `numGamesPlayed` is
implicitly `const`
    constexpr int32_t const& ngp = numGamesPlayed;

    // Constant array
    // Implicitly `const` with `const` elements
    constexpr int32_t exponentialBackoffDelays[4] = {
100, 200, 400, 800 };
}
```

So far we've just used primitives, references to primitives, and arrays of primitives. Classes are allowed too, but with a few restrictions. First, they must be either an aggregate, a lambda, or have at least one non-copy, non-move `constexpr` constructor. We'll get into `constexpr` functions in the next section.

```
struct NonLiteralType
{
    // Delete the copy and move constructors
    NonLiteralType(const NonLiteralType&) = delete;
    NonLiteralType(const NonLiteralType&&) = delete;
private:
    // Add a private non-static data member so it's not
an aggregate
    int32_t Val;
};
```

```
// Compiler error: not a "literal type"
constexpr NonLiteralType nlt{};
```

Second, it must have a `constexpr` destructor.

```
struct NonLiteralType
{
    // Destructor isn't constexpr
    NonLiteralType()
    {
    }
};

// Compiler error: NonLiteralType doesn't have a
constexpr destructor
constexpr NonLiteralType nlt{};

struct LiteralTypeA
{
    // Explicitly compiler-generated destructor is
constexpr
    LiteralTypeA() = default;
};

struct LiteralTypeB
{
    // Implicitly compiler-generated destructor is
constexpr
```

```
};

// OK
constexpr LiteralTypeA lta{};
constexpr LiteralTypeB ltb{};
```

Third, if it's a `union` then at least one of its non-`static` data members must be a "literal type." If it's not a `union`, *all* of its non-`static` data members and all the data members of its base classes must be "literal types."

```
union NonLiteralUnion
{
    NonLiteralType nlt1;
    NonLiteralType nlt2;
};

// Compiler error: all of the union's non-static data
members are non-literal
constexpr NonLiteralUnion nlu{};

struct NonLiteralStruct
{
    NonLiteralType nlt1;
    int32_t Val; // Primitives are literal types
};

// Compiler error: not all of the struct's non-static
```

```
data members are literal
constexpr NonLiteralStruct nls{};
```

If we satisfy these requirements, we're free to make constant class variables. Here's a simple aggregate:

```
struct Vector2
{
    float X;
    float Y;
};

// Constant class instance
constexpr Vector2 ORIGIN{0, 0};
```

One last thing to keep in mind with `constexpr` variables: they're incompatible with `constinit` variables. This C++20 keyword is used to require that a variable be constant-initialized:

```
// Requires `ok` to be constant-initialized
constinit const char* ok = "OK";

// Compiler error: can't be both constexpr and constinit
consteval constinit const char* ok2 = "OK";

// Compiler error: not initialized with a constant
constinit const char* err = rand() == 0 ? "f" : "t";
```

Constant Functions

Functions in C++ may also be `constexpr`. This means they can be executed at compile time:

```
constexpr int32_t SumOfFirstN(int32_t n)
{
    int32_t sum = 0;
    for (int32_t i = 1; i <= n; ++i)
    {
        sum += i;
    }
    return sum;
}

// 1 + 2 + 3
DebugLog(SumOfFirstN(3)); // 6
```

Because `3` is a compile-time constant, the call to `SumOfFirstN(3)` will be executed during compilation and replaced by the return value to become:

```
DebugLog(6); // 6
```

If the argument to `SumOfFirstN` *isn't* a compile-time constant, `SumOfFirstN` will be compiled and called like normal:

```
// Read `n` from a file
FILE* handle = fopen("/path/to/file", "r");
int32_t n{0};
fread(&n, sizeof(n), 1, handle);

// Argument not known at compile time
// Depends on the state of the file system
// SumOfFirstN call happens at runtime
DebugLog(SumOfFirstN(n)); // 6

fclose(handle);
```

If we don't want the function to be callable at runtime, such as to avoid accidentally increasing the executable size or performing compile-time computation at runtime, C++20 allows us to replace `constexpr` with `consteval`. The only difference between `constexpr` and `consteval` is that we'll get a compiler error if we try to call a `consteval` function at runtime.

```
consteval int32_t SumOfFirstN(int32_t n)
{
    int32_t sum = 0;
    for (int32_t i = 1; i <= n; ++i)
    {
        sum += i;
    }
    return sum;
}
```

In order to be `constexpr`, functions must meet certain criteria. Since their introduction in C++11, this criteria has been greatly relaxed in C++14 and C++20. Future versions of the language will likely continue the trend. For now, we'll just look at the rules for C++20.

First, all of their parameters and their return type must be "literal types."

```
// Not a literal type due to non-constexpr destructor
struct NonLiteral
{
    ~NonLiteral()
    {
    }
};

// Compiler error: constexpr function parameters must be
literal types
constexpr void Foo(NonLiteral nl)
{
}

// Compiler error: constexpr function return value must
be a literal type
constexpr NonLiteral Goo()
{
    return {};
}
```

Second, a `constexpr` constructor or destructor can't be in a class that has virtual base classes or non-`constexpr` base class constructors or destructors.

```
struct NonLiteralBase
{
};

struct NonLiteralCtor : virtual NonLiteralBase
{
    // Compiler error: constructor can't be constexpr
with virtual base classes
    constexpr NonLiteralCtor()
    {
    }
};

struct NonLiteralDtor : virtual NonLiteralBase
{
    // Compiler error: destructor can't be constexpr with
virtual base classes
    constexpr ~NonLiteralDtor()
    {
    }
};
```

Third, the body of the function can't have any `goto` statements or labels except `case` and `default` in `switch` statements:

```
constexpr int32_t GotoLoop(int32_t n)
{
    int32_t sum = 0;
    int32_t i = 1;

    // Compiler error: constexpr function can't have non-
case, non-default label
beginLoop:

    if (i > n)
    {
        // Compiler error: constexpr function can't have
goto
        goto endLoop;
    }
    sum += i;

    // Compiler error: constexpr function can't have non-
case, non-default label
endLoop:
    return sum;
}
```

Fourth, all local variables have to be “literal types:”

```
constexpr void Foo()
{
```

```
    // Compiler error: constexpr function can't have non-
literal variables
    NonLiteral nl{};
}
```

Fifth, the function can't have any `static` variables:

```
constexpr int32_t GetNextInt()
{
    // Compiler error: constexpr functions can't have
static variables
    static int32_t next = 0;

    return ++next;
}
```

Any function that follows all these rules is free to be `constexpr` or `consteval`.

Constant If

Since C++17, a new form of `if` is available: `if constexpr (...)`. These are evaluated at compile time regardless of whether they're in a `constexpr` function or not. The compiler then removes the `if` in favor of either the code in the `if` block or the `else` block. They're very useful for removing code from the executable to reduce its size and to remove the cost of branch instructions at runtime.

For example, say we want to set the minimum severity level that is logged at compile time and not have to check it every time a log message is written. We could use a compiler-defined preprocessor symbol (`LOG_LEVEL`), a `constexpr` string equality function (`IsStrEqual`), and `if constexpr` to either log or not log:

```
// The compiler sets a preprocessor symbol based on its
configuration
// It's like we wrote this into the C++ code:
#define LOG_LEVEL "WARN"

// Constant function that compares NUL-terminated strings
for exact equality
constexpr bool IsStrEqual(const char* str, const char*
match)
{
    for (int i = 0; ; ++i)
    {
        char s = str[i];
        char m = match[i];
        if (s)
```

```
        {
            if (m)
            {
                if (s != m)
                {
                    return false;
                }
            }
            else
            {
                return false;
            }
        }
        else
        {
            return !m;
        }
    }
    return true;
}

// Logs debug messages if LOG_LEVEL is DEBUG
void LogDebug(const char* msg)
{
    // Compare LOG_LEVEL to "DEBUG" at compile time
    // If false, the WriteLog call is removed from the
executable
    if constexpr (IsStrEqual(LOG_LEVEL, "DEBUG"))
```

```
    {
        WriteLog("DEBUG", msg);
    }
}

// Logs warning messages if LOG_LEVEL is DEBUG or WARN
void LogWarn(const char* msg)
{
    // Compare LOG_LEVEL to "DEBUG" and "WARN" at compile
time
    // If false, the WriteLog call is removed from the
executable
    if constexpr (IsStrEqual(LOG_LEVEL, "DEBUG") ||
        IsStrEqual(LOG_LEVEL, "WARN"))
    {
        WriteLog("WARN", msg);
    }
}

// Logs warning messages if LOG_LEVEL is DEBUG, WARN, or
ERROR
void LogError(const char* msg)
{
    // Compare LOG_LEVEL to "DEBUG", "WARN", and "ERROR"
at compile time
    // If false, the WriteLog call is removed from the
executable
    if constexpr (IsStrEqual(LOG_LEVEL, "DEBUG") ||
```

```
        IsStrEqual(LOG_LEVEL, "WARN") ||
        IsStrEqual(LOG_LEVEL, "ERROR"))
    {
        WriteLog("ERROR", msg);
    }
}
```

Static Assertions

Similar to `if constexpr`, we can write assertions that execute at compile time rather than run time. For example, we could `static_assert` to make sure the `LOG_LEVEL` was set to a valid value. If it's not the code won't compile:

```
static_assert(IsStrEqual(LOG_LEVEL, "DEBUG") ||
    IsStrEqual(LOG_LEVEL, "WARN") ||
    IsStrEqual(LOG_LEVEL, "ERROR"),
    "Invalid log level: " LOG_LEVEL);
```

We could also check to make sure the code is being compiled in a supported build environment:

```
static_assert(MSC_VER >= 1900, "Only Visual Studio 2015+
is supported");
static_assert(_WIN64, "Only 64-bit Windows is
supported");
```

Or, as is more traditional for assertions, we could make sure that we didn't make a mistake in our code. For example, it's common to define a `struct` that's serialized and deserialized when reading from or writing to files or network sockets. We can use `static_assert` to make sure it's the right size and we haven't inadvertently increased it by adding data members or padding:

```
struct PlayerUpdatePacket
{
```

```
    int32_t PlayerId;
    float PositionX;
    float PositionY;
    float PositionZ;
    float VelocityX;
    float VelocityY;
    float VelocityZ;
    int32_t NumLives;
};
static_assert(
    sizeof(PlayerUpdatePacket) == 32,
    "PlayerUpdatePacket serializes to wrong size");
```

Since C++17, we can omit the error message if we don't think it's helpful:

```
static_assert(sizeof(PlayerUpdatePacket) == 32);
```

Constant Expressions

Now that we have a wealth of compile-time features in `constexpr` variables, `constexpr` functions, `if constexpr`, and `static_assert`, we need to know when we're allowed to use these and when we're not. Clearly literals like `32` are constant expressions and calls to `rand()` to get a random number are not, but many cases are not so clear-cut.

C++ qualifies an expression as constant as long as it *doesn't* match any of a list of criteria. Since C++11, each version of the language has relaxed these restrictions. The language is broadly moving toward being entirely executable at compile time. Until then, we need to know what's *not* allowed in a compile-time expression so we know what we can't use with all these compile-time features. Let's go through the rules as of C++20.

First, the `this` pointer can't be used, implicitly or explicitly, outside of `constexpr` member functions and `constexpr` constructors:

```
struct Test
{
    int32_t Val{123};

    int32_t GetVal()
    {
        // Explicitly use the this pointer
        // Compiler error: can't use `this` outside of
constexpr member function
        constexpr int32_t val = this->Val;
```

```
        return val;
    }
};
```

The same goes for lambdas referencing captured objects, since they are effectively accessed via the `this` pointer to the lambda class:

```
const int32_t outside = 123;
auto lambda = [outside]
{
    // Compiler error: can't reference variables outside
the lambda
    constexpr int32_t const* pOutside = &outside;

    DebugLog(*pOutside);
};
lambda();
```

Second, calls to non-`constexpr` functions and constructors aren't allowed:

```
struct Test
{
    int32_t Val{123};

    // User-defined conversion operator to int32_t
    operator int32_t()
    {
```

```
        return Val;
    }
};

// Try to make a Test and convert it to an int32_t
// Compiler error: call to Test constructor which isn't
constexpr
constexpr int32_t val = Test{};
```

Third, we can't make calls to `constexpr` functions that are only declared but not yet defined:

```
// Declare constexpr function
constexpr int32_t Pow(int32_t x, int32_t y);

// Compiler error: Pow isn't defined yet
constexpr int32_t eight = Pow(2, 3);

// Define constexpr function
constexpr int32_t Pow(int32_t x, int32_t y)
{
    int32_t result = x;
    for (; y > 1; --y)
    {
        result *= x;
    }
    return result;
}
```

Fourth, calls to `constexpr` `virtual` functions of classes that aren't "literal types" and whose lifetimes began before the expression:

```
struct Base
{
    constexpr virtual int32_t GetVal()
    {
        return 1;
    }
};

struct Derived : Base
{
    constexpr virtual int32_t GetVal() override
    {
        return 123;
    }
};

// Class begins lifetime before the constant expression
Derived d{};

// Compiler error: can't call constexpr virtual function
on class that began its
// lifetime before the constant expression
constexpr int32_t val = d.GetVal();

// OK: Derived object begins lifetime during constant
```

```
expression
constexpr int32_t val2 = Derived{}.GetVal();
```

Fifth, triggering any form of undefined behavior. This rule adds a safety net to `constexpr` code. Undefined behavior simply won't compile.

```
// Compiler error: dividing by zero is undefined behavior
constexpr float f = 3.14f / 0.0f;

// Compiler error: signed integer overflow is undefined
behavior
constexpr int32_t i = 0x7fffffff + 1;
```

Sixth, lvalues can't be used as rvalues if the lvalue isn't allowed in a constant expression. For example, a non-`const` lvalue isn't allowed but a `const` lvalue is:

```
// i1 can't be used in a constant expression because it's
not const
int32_t i1 = 123;

// i2 can be used in a constant expression because it is
const
const int32_t i2 = 123;

// i3 can't be used in a constant expression because it's
not initialized with
```

```
// a constant expression
const int32_t i3 = i1;

// Compiler error: i1 can't be used
constexpr int32_t i4 = i1;

// Compiler error: i3 can't be used
constexpr int32_t i5 = i3;

// OK: i2 can be used
constexpr int32_t i6 = i2;
```

References are not an allowed workaround since they are just a synonym for an object:

```
struct HasVal
{
    int32_t Val{123};
};

HasVal hv{};
constexpr HasVal const& rhv{hv};
constexpr int32_t ri{rhv.Val};

constexpr HasVal hv2{};
constexpr HasVal const& rhv2{hv2};
constexpr int32_t ri2{rhv2.Val};
```

Seventh, only the active member of a union can be used. Accessing non-active members is a form of undefined behavior that's commonly allowed by compilers in run-time code but disallowed in compile-time code.

```
union IntFloat
{
    int32_t Int;
    float Float;

    constexpr IntFloat()
    {
        // Make Int the active member
        Int = 123;
    }
};


// Call constexpr constructor which makes Int active
constexpr IntFloat u{};

// OK: can use Int because it's the active member
constexpr int32_t i = u.Int;

// Compiler error: can't use Float because it's not the
active member
constexpr float f = u.Float;
```

The compiler-generated copy or move constructor or assignment operator of a `union` whose active member is `mutable` is also disallowed:

```
union IntFloat
{
    mutable int32_t Int;
    float Float;

    constexpr IntFloat()
    {
        Int = 123;
    }
};

constexpr IntFloat u{};
constexpr IntFloat u2{u};
```

Eighth, converting from `void*` to any other pointer type:

```
constexpr int32_t i = 123;
constexpr void const* pv = &i;

// Compiler error: constant expressions can't convert
void*
constexpr int32_t const* pi = static_cast<int32_t const*>
(pv);
```

Ninth, any use of reinterpret_cast:

```
constexpr int32_t i = 123;

// Compiler error: constant expressions can't use
reinterpret_cast even to cast
//                  to the same type
constexpr int32_t i2 = reinterpret_cast<int32_t>(i);
```

Tenth, modifying an object that's not a literal type whose lifetime began within the constant expression:

```
constexpr int Foo(int val)
{
    // val begins its liftime here, before the constant
expression starts

    // Compiler error: constant expression can't modify
object whose lifetime
    // started before
    constexpr int ret = ++val;

    return ret;
}
```

Eleventh, `new` and `delete` unless the allocated memory is deleted by the constant expression:

```
// Compiler error: memory allocated by `new` not deleted
in constant expression
constexpr int32_t* pi = new int;

constexpr int32_t GetInt()
{
    int32_t* p = new int32_t;
    *p = 123;
    int32_t ret = *p;
    delete p;
    return ret;
}

// OK: GetInt() call deletes memory it allocated with the
new operator
constexpr int32_t i = GetInt();
```

Twelfth, comparing pointers is technically “unspecified behavior” and not allowed in a constant expression:

```
int32_t i = 123;
int32_t* pi1 = &i;
int32_t* pi2 = &i;

// Compiler error: can’t compare pointers in a constant
expression
constexpr bool b = pi1 < pi2;
```

Thirteenth, while we can catch exceptions, we can't throw them:

```
constexpr void Explode()
{
    // Compiler error: can't throw in a constant
expression
    throw "boom";
}
```

dynamic_cast and typeid are also not allowed to throw an exception:

```
struct Combatant
{
    virtual int32_t GetMaxHealth() = 0;
};

struct Enemy : Combatant
{
    virtual int32_t GetMaxHealth() override
    {
        return 100;
    }
};

struct TutorialEnemy : Combatant
{
    virtual int32_t GetMaxHealth() override
    {
```

```
        return 10;
    }
};

Enemy e;
Combatant& c{e};

// Compiler error: can't call dynamic_cast in a constant
expression if it
// will throw an exception
constexpr TutorialEnemy& te{dynamic_cast<TutorialEnemy&>
(c)};

Enemy* p = nullptr;

// Compiler error: can't call typeid in a constant
expression if it will
// throw an exception
constexpr auto name = typeid(p).name();
```

Fourteenth, and finally, variadic functions using `va_start`, `va_arg`, and `va_end` which are generally not recommended to be used anyways:

```
constexpr void DebugLogAll(int count, ...)
{
    va_list args;
```

```
    // Compiler error: can't use va_start in constant
expressions
    va_start(args, count);

    for (int i = 0; i < count; ++i)
    {
        const char* msg = va_arg(args, const char*);
        DebugLog(msg);
    }

    va_end(args);
}
```

Conclusion

C++ provides a powerful set of compile-time programming options with `constexpr` variables, `constexpr` functions, `constexpr if`, and `static_assert`. Each new version of the language makes more and more of the regular, run-time features available at compile time. While there are still quite a few restrictions, many of them are for esoteric features like variadic functions or to explicitly prevent bugs such as by forbidding undefined behavior.

The general direction of the language is to eventually be fully available at compile time so there's no need to use another language or write another program to generate source code. The result is a uniform build process and the de-duplication of compile-time and run-time code. By simply adding the `constexpr` keyword, we can often move run-time calculations to compile time and increase run-time performance by using those pre-computed results.

In contrast, C# support for compile-time programming is limited to using built-in operators on primitives and `string` in `const` and default function arguments. These have been available since the first version (in 2002) and no more features have been added or announced since then. Compile-time programming is virtually always done by an external tool and build step that generates `.cs` or `.dll` files. This seems to represent a philosophical difference between the two languages.

24. Preprocessor

Conditionals

Just like in C#, the C++ preprocessor runs at an early stage of compilation. This is after the bytes of the file are interpreted as characters and comments are removed, but before the main compilation of language concepts like variables and functions. The preprocessor therefore has a very limited understanding of the source code.

It takes this limited understanding of the source code and makes textual substitutions to it. When it's done, the resulting source code is compiled.

One common use of this in C# are the conditional "directives:" `#if`, `#else`, `#elif`, and `#endif`. These allow branching logic to take place during the preprocessing step of compilation. C# allows for logic on boolean preprocessor symbols:

```
// C#
static void Assert(bool condition)
{
    #if DEBUG && (ASSERTIONS_ENABLED == true)
        if (!condition)
        {
            throw new Exception("Assertion failed");
        }
    #endif
}
```

If the `#if` expression evaluates to `false` then the code between the `#if` and `#endif` is removed:

```
// C#
static void Assert(bool condition)
{
}
```

This helps us reduce the size of the generated executable and improve run-time performance by removing instructions and memory accesses.

One common mistake is to assume the preprocessor understands more about the structure of the source code than it really does. For example, we might assume that it understands what identifiers are:

```
// C#
void Foo()
{
    #if Foo
        DebugLog("Foo exists");
    #else
        DebugLog("Foo does not exist"); // Gets printed
    #endif
}
```

C++ has similar support for preprocessor conditionals. They're even named `#if`, `#else`, `#elif`, and `#endif`. The above C# examples are actually valid C++!

The two languages differ in a few minor ways. First, `#if ABC` in C++ checks the *value* of `ABC`, not just whether it's defined.

```
// Assume the value of ZERO is 0
#if ZERO
    DebugLog("zero");
#else
    DebugLog("non-zero"); // Gets printed
#endif
```

There are a couple ways to avoid this. First, we can use the preprocessor `defined` operator to check whether the symbol is defined instead of checking its value:

```
#if defined(ZERO) // evaluates to 1, which is true
    DebugLog("zero"); // Gets printed
#else
    DebugLog("non-zero");
#endif

// Alternate version without parentheses
#if defined ZERO // evaluates to 1, which is true
    DebugLog("zero"); // Gets printed
#else
    DebugLog("non-zero");
#endif
```

The other way is to use `#ifdef` and `#ifndef` instead of of `#if`:

```
#ifdef ZERO // ZERO is defined. Its value is irrelevant.
    DebugLog("zero"); // Gets printed
#else
    DebugLog("non-zero");
#endif

#ifndef ZERO // Check if NOT defined
    DebugLog("zero");
#else
    DebugLog("non-zero"); // Gets printed
#endif
```

These checks are commonly used to implement header guards. Since C++17, they can also be used with `__has_include` which evaluates to `1` if a header exists and `0` if it doesn't. This is often used to check whether optional libraries are available or to choose from one of several equivalent libraries:

```
// If the system provides DebugLog via the debug_log.h
header
#if __has_include(<debug_log.h>)
    // Use the system-provided DebugLog
    #include <debug_log.h>
// The system does not provide DebugLog
#else
    // Define our own version using puts from the C
Standard Library
    #include <cstdio>
```

```
    void DebugLog(const char* message)
    {
        puts(message);
    }
#endif
```

This `__has_include(<header_name>)` check uses the same header file search that `#include <header_name>` would. To check the header file search of `#include "header_name"`, we can use `__has_include("header_name")`.

Macros

Both languages allow defining preprocessor symbols with `#define` and un-defining them with `#undef`:

```
// Define a preprocessor symbol
#define ENABLE_LOGGING

void LogError(const char* message)
{
    // Check if the preprocessing symbol is defined
    // It is, so the DebugLog remains
    #ifdef ENABLE_LOGGING
        DebugLog("ERROR", message);
    #endif
}

// Un-define the preprocessor symbol
#undef ENABLE_LOGGING

void LogTrace(const char* message)
{
    // Check if the preprocessing symbol is defined
    // It isn't, so the DebugLog is removed
    #ifdef ENABLE_LOGGING
        DebugLog("TRACE", message);
    #endif
}
```

```
void Foo()
{
    LogError("whoops"); // Prints "ERROR whoops"
    LogTrace("got here"); // Nothing printed
}
```

C# requires `#define` and `#undef` to appear only at the top of the file, but C++ allows them anywhere.

C++ also goes *way* beyond these simple preprocessor symbol definitions. It has a full "macro" system that allows for textual substitution. While this is generally discouraged in "Modern C++," its use is still ubiquitous for certain tasks. Sometimes it's used when the language doesn't provide a viable alternative or at least didn't when the code was written. Regardless, macros are widely used and it's important to know how they work.

First, we can define an "object-like" macro by providing a value to the preprocessor symbol. Unlike C#, the value doesn't have to be a boolean:

```
// Define an object-like macros
#define LOG_LEVEL 1
#define LOG_LEVEL_ERROR 3
#define LOG_LEVEL_WARNING 2
#define LOG_LEVEL_DEBUG 1

void LogWarning(const char* message)
{
    // The preprocessor symbol can be used in #if
```

```
expressions
    #if LOG_LEVEL <= LOG_LEVEL_WARNING
        // The preprocessor symbol will be replaced with
its value
        DebugLog(LOG_LEVEL_WARNING, message);

        // After preprocessing, the previous line
becomes:
        DebugLog(2, message);
    #endif
}
```

We can also define “function-like” macros that take parameters:

```
// Define a function-like macro
#define MADD(x, y, z) x*y + z

void Foo()
{
    int32_t x = 2;
    int32_t y = 3;
    int32_t z = 4;

    // Call the function-like macro
    int32_t result = MADD(x, y, z);

    // After preprocessing, the previous line becomes:
    int32_t result = x*y + z;
```

```
    DebugLog(result); // 10
}
```

Unlike a runtime function call, calling a function-like macro simply performs textual substitution. It's easy to forget this, especially when the macro is named like a normal function. This can lead to bugs and performance problems because argument expressions aren't evaluated before the macro is called:

```
// Function-like macro named like a normal function, not
ALL_CAPS
#define square(x) x*x

int32_t SumOfRandomNumbers(int32_t n)
{
    int32_t sum = 0;
    for (int32_t i = 0; i < n; ++i)
    {
        sum += rand();
    }
    return sum;
}

void Foo()
{
    // Call a very expensive function
    int32_t result = square(SumOfRandomNumbers(1000000));
```

```
    // After preprocessing, the previous line becomes:
    int32_t result =
SumOfRandomNumbers(1000000)*SumOfRandomNumbers(1000000);

    DebugLog(result); // {some random number}
}
```

With a normal function call, `SumOfRandomNumbers(1000000)` would be evaluated before the function is called. With macros, it's just textually replaced so `square` ends up making two calls to it. The call is very expensive, so we have a performance problem. It's also a bug because we're no longer necessarily multiplying the same number by itself since the two calls may return different numbers.

To see more clearly how bugs arise, consider this macro call:

```
void Foo()
{
    int32_t i = 1;
    int32_t result = square(++i);

    // After preprocessing, the previous line becomes:
    int32_t result = ++i*++i;

    DebugLog(result, i); // 6, 3
}
```

Again, the argument (`++i`) isn't evaluated before the macro call but rather just repeated every time the macro refers to the parameter. This means `i` is incremented from `1` to `2` then again to `3` before the

multiplication (`*`) produces the result of `2*3=6` and sets `i` to `3`. If this were a function call, we'd expect `2*2=4` and for the value of `i` to be `2` afterward. These potential bugs are one reason why macros are discouraged.

Function-like macros have access to a couple of special operators: `#` and `##`. The `#` operator wraps an argument in quotes to create a string literal:

```
// Wrap msg in quotes to create "msg"
#define LOG_TIMESTAMPED(msg) DebugLog(GetTimestamp(),
#msg);

void Foo()
{
    // No need for quotes. hello becomes "hello".
    LOG_TIMESTAMPED(hello) // {timestamp} hello

    // Extra quotes are added and existing quotes are
escaped: ""hello""
    LOG_TIMESTAMPED("hello") // {timestamp} "hello"
}
```

The `##` operator is used to concatenate two symbols, which may be arguments:

```
// Each line concatenates some literal text (e.g. m_)
with the value of name
// Backslashes are used to make a multi-line macro
```

```
#define PROP(type, name) \
    private: type m_##name; \
    public: type Get##name() const { return m_##name; } \
    public: void Set##name(const type & val) { m_##name =
val; }

struct Vector2
{
    PROP(float, X)
    PROP(float, Y)

    // These macro calls are replaced with:

    private: float m_X;
    public: float GetX() const { return m_X; }
    public: void SetX(const float & val) { m_X = val; }
    private: float m_Y;
    public: float GetY() const { return m_Y; }
    public: void SetY(const float & val) { m_Y = val; }
};

void Foo()
{
    Vector2 vec;
    vec.SetX(2);
    vec.SetY(4);
    DebugLog(vec.GetX(), vec.GetY()); // 2, 4
}
```

Macros may also take a variable number of parameters using `...` similar to functions. `__VA_ARGS__` is used to access the arguments:

```
#define LOG_TIMESTAMPED(level, ...) DebugLog(level,
GetTimestamp(), __VA_ARGS__);

void Foo()
{
    LOG_TIMESTAMPED("DEBUG", "hello", "world") // DEBUG
{timestamp} hello world

    // This macro call is replaced by:

    DebugLog("DEBUG", GetTimestamp(), "hello", "world");
}
```

In C++20, `__VA_OPT__(x)` is also available. If `__VA_ARGS__` is empty, it's replaced by nothing. If `__VA_ARGS__` *isn't* empty, it's replaced by `x`. This can be used to make parameters in macros like `LOG_TIMESTAMPED` optional:

```
// __VA_OPT__(,) adds a comma only if __VA_ARGS__ isn't
empty, meaning the
// caller passed some log messages
#define LOG_TIMESTAMPED(...) DebugLog(GetTimestamp()
__VA_OPT__(,) __VA_ARGS__);

void Foo()
```

```
{
    LOG_TIMESTAMPED() // {timestamp}
    LOG_TIMESTAMPED("hello", "world") // {timestamp}
hello world

    // These macro calls are replaced by:

    DebugLog(GetTimestamp()  );

    DebugLog(GetTimestamp() , "hello", "world");
}
```

Without `__VA_OPT__`, we wouldn't know if the macro should put a `,` or not because we wouldn't know if there are any arguments to pass after it.

Built-in Macros and Feature-Testing

Just like how C# pre-defines the `DEBUG` and `TRACE` preprocessor symbols, C++ pre-defines some object-like macros:

Name	Value	Meaning
`__cplusplus`	`199711L` (C++98 and C++03) `201103L` (C++11) `201402L` (C++14) `201703L` (C++17) `202002L` (C++20)	C++ language version
`__STDC_HOSTED__`	`1` if there is an OS, `0` if not	
`__FILE__`	`"mycode.cpp"`	Name of the current file
`__LINE__`	`38`	Current line number
`__DATE__`	`"2020 10 26"`	Date the code was compiled
`__TIME__`	`"02:00:00"`	Time the code was compiled

Name	Value	Meaning
`__STDCPP_DEFAULT_NEW_ALIGNMENT__`	`8`	Default alignment of `new`. Only in C++17 and up.

Since C++20, there are a ton of "feature test" macros available in the `<version>` header file. These are all object-like and their values are the date that the language or Standard Library feature was added to C++. The intention is to compare them to `__cplusplus` to determine whether the feature is supported or not. There are way too many to list here, but the following shows a couple in action:

```
void Foo()
{
    if (__cplusplus >= __cpp_char8_t)
    {
        DebugLog("char8_t is supported in the language");
    }
    else
    {
        DebugLog("char8_t is NOT supported in the
language");
    }

    if (__cplusplus >= __cpp_lib_byte)
    {
        DebugLog("std::byte is supported in the Standard
```

```
Library");
    }
    else
    {
        DebugLog("std::byte is NOT supported in the
Standard Library");
    }
}
```

A complete list is available in the C++ Standard's definition of the `<version>` header file.

Miscellaneous Directives

The pre-defined `__FILE__` and `__LINE__` values can be overridden by another preprocessor directive: `#line`. This works just like in C# except that `default` and `hidden` aren't allowed:

```
void Foo()
{
    DebugLog(__FILE__, __LINE__); // main.cpp, 38
#line 100
    DebugLog(__FILE__, __LINE__); // main.cpp, 100
#line 200 "custom.cpp"
    DebugLog(__FILE__, __LINE__); // custom.cpp, 200
}
```

`#error` can be used to make the compiler produce an error:

```
#ifndef _MSC_VER
    #error Only Visual Studio is supported
#endif
```

`#pragma` is used to allow compilers to provide their own preprocessor directives, just like in C#:

```
// mathutils.h

// Compiler-specific alternative to header guards
```

```
#pragma once

float SqrMagnitude(const Vector2& vec)
{
    return vec.X*vec.X + vec.Y*vec.Y;
}
```

`_Pragma("expr")` can be used instead of `#pragma expr`. It has exactly the same effect:

```
_Pragma("once")
```

C#'s `#region` and `#endregion` aren't supported in C++, but compilers like Visual Studio allow it via `#pragma`:

```
#pragma region Math

float SqrMagnitude(const Vector2& vec);
float Dot(const Vector2& a, const Vector2& b);

#pragma endregion Math
```

Usage and Alternatives

Each new version of C++ makes usage of the preprocessor less necessary. For example, C++11 introduced constexpr variables which removed a lot of the reasons to use object-like macros:

```
// Before C++11
#define PI 3.14f

// After C++11
constexpr float PI = 3.14f;
```

This made `PI` an actual object so it has a type (`float`), its address can be taken (`&PI`), and just generally used like other objects rather than as a textually-replaced `float` literal. The benefits become much greater with `struct` types, lambda classes, and other non-primitives where it's not really possible to make a macro for general use:

```
// Before C++11
// This isn't usable in many contexts like
Foo(EXPONENTIAL_BACKOFF_TIMES)
#define EXPONENTIAL_BACKOFF_TIMES { 1000, 2000, 4000,
8000, 16000 }

// After C++11
// This works like any array object:
constexpr int32_t ExponentialBackoffTimes[] = { 1000,
2000, 4000, 8000, 16000 };
```

Likewise, `constexpr` and `consteval` functions have removed a lot of the need for function-like macros:

```
constexpr int32_t Square(int32_t x)
{
    return x * x;
}

void Foo()
{
    int32_t i = 1;
    int32_t result = Square(++i);
    DebugLog(result); // 4
}
```

These behave like regular functions rather than textual substitution. We skip all the bugs and performance problems that macros might cause but keep the compile-time evaluation. We can even force compile-time evaluation in C++20 with `consteval`. We get strong typing, so `Square("FOO")` is an error. We can use the function at run-time, not just compile time. It behaves like any other function: we can take function pointers, we can create member functions, and so forth.

Still, macros provide a sort of escape hatch for when we simply can't express something without raw textual substitution. The `PROP` macro example above generates members with access specifiers. There's no way to do that otherwise. That example might not be the best idea, but others really are. A classic example is an assertion macro:

```
// When assertions are enabled, define ASSERT as a macro
that tests a boolean
// and logs and terminates the program when it's false.
#ifdef ENABLE_ASSERTS
    #define ASSERT(x) \
        if (!(x)) \
        { \
            DebugLog("assertion failed"); \
            std::terminate(); \
        }
// When assertions are disabled, assert does nothing
#else
    #define ASSERT(x)
#endif

bool IsSorted(const float* vals, int32_t length)
{
    for (int32_t i = 1; i < length; ++i)
    {
        if (vals[i] < vals[i-1])
        {
            return false;
        }
    }
    return true;
}

float GetMedian(const float* vals, int32_t length)
```

```
{
    ASSERT(vals != nullptr);
    ASSERT(length > 0);
    ASSERT(IsSorted(vals, length));
    if ((length & 1) == 1)
    {
        return vals[length / 2]; // odd
    }
    float a = vals[length / 2 - 1];
    float b = vals[length / 2];
    return (a + b) / 2;
}

void Foo()
{
    float oddVals[] = { 1, 3, 3, 6, 7, 8, 9 };
    DebugLog(GetMedian(oddVals, 7));

    float evenVals[] = { 1, 2, 3, 4, 5, 6, 8, 9 };
    DebugLog(GetMedian(evenVals, 8));

    DebugLog(GetMedian(nullptr, 1));

    float emptyVals[] = {};
    DebugLog(GetMedian(emptyVals, 0));

    float notSortedVals[] = { 3, 2, 1 };
```

```
    DebugLog(GetMedian(notSortedVals, 3));
}
```

Calling `ASSERT` with assertions enabled performs the following replacement:

```
ASSERT(IsSorted(vals, length));

// Becomes:

if (!(IsSorted(vals, length)))
{
    DebugLog("assertion failed");
    std::terminate();
}
```

When disabled, everything's removed including the expressions passed as arguments:

```
ASSERT(IsSorted(vals, length));

// Becomes:
```

Now imagine we had used a `constexpr` function instead of a macro:

```
#ifdef ENABLE_ASSERTS
    constexpr void ASSERT(bool x)
```

```
    {
        if (!x)
        {
            DebugLog("assertion failed");
            std::terminate();
        }
    }
#else
    constexpr void ASSERT(bool x)
    {
    }
#endif
```

When assertions are disabled, we get the empty `constexpr` function:

```
constexpr void ASSERT(bool x)
{
}
```

But when we call `ASSERT` the arguments still need to be evaluated even though the function itself does nothing:

```
ASSERT(IsSorted(vals, length));

// Is equivalent to:

bool x = IsSorted(vals, length);
Assert(x); // does nothing
```

The compiler *might* be able to determine that the call to `IsSorted` has no side effects and can be safely removed. In many cases, it won't be able to make this determination and an expensive call to `IsSorted` will still take place. We don't want this to happen, so we use a macro.

Macros can also be used to implement a primitive form of C# generics or C++ templates, which we'll cover soon in the book:

```
// "Generic"/"template" of a Vector2 class
#define DEFINE_VECTOR2(name, type) \
    struct name \
    { \
        type X; \
        type Y; \
    };

// Invoke the macro to generate Vector2 classes
DEFINE_VECTOR2(Vector2f, float);
DEFINE_VECTOR2(Vector2d, double);

// "Generic"/"template" of a function
#define DEFINE_MADD(type) \
    type Madd(type x, type y, type z) \
    { \
        return x*y + z; \
    }

```

```
// Invoke the macro to generate Madd functions
DEFINE_MADD(float);
DEFINE_MADD(int32_t);

void Foo()
{
    // Use the generated Vector2 classes
    // Use sizeof to show that they have different component sizes
    Vector2f v2f{2, 4};
    DebugLog(sizeof(v2f), v2f.X, v2f.Y); // 8, 2, 4

    Vector2d v2d{20, 40};
    DebugLog(sizeof(v2d), v2d.X, v2d.Y); // 16, 20, 40

    // Use the generated Madd functions
    // Use typeid on the return value to show that they're overloads
    float xf{2}, yf{3}, zf{4};
    auto maddf{Madd(xf, yf, zf)};
    DebugLog(typeid(maddf) == typeid(float)); // true
    DebugLog(typeid(maddf) == typeid(int32_t)); // false

    int32_t xi{2}, yi{3}, zi{4};
    auto maddi{Madd(xi, yi, zi)};
    DebugLog(typeid(maddi) == typeid(float)); // false
    DebugLog(typeid(maddi) == typeid(int32_t)); // true
}
```

This form of code generation is commonly used in C codebases that lack C++ templates. When templates are available, as they are in all versions of C++, they are the preferred option for many reasons. One reason is the ability to "overload" a class name so we just have `Vector2` rather than coming up with awkward unique names like `Vector2f` and `Vector2d`.

Another is that there's no need for, usually large, lists of `DEFINE_X` macro calls for every permutation of types needed in every class and function. This really gets out of control when there are several "type parameters." Instead, the compiler generates all the permutations of the class or function based on our usage of them so we don't need to explicitly maintain such lists.

There are many more reasons that we'll get into when we cover templates later in the book.

Conclusion

The two languages have a lot of overlap in their use of the preprocessor. It runs at the same stage of compilation and features many identically-named directives with the same functionality.

The major points of divergence are in `#include`, an essential part of the build model before C++20, and in macros created by `#define`. Function-like macros represent another form of compile-time programming that runs during preprocessing as opposed to `constexpr` which runs during main compilation. They're also another form of generics or templates. While their necessity has diminished over time, they are still essential for some tasks and convenient for others.

25. Intro to Templates

What are Templates?

Templates really are what their name suggests: a template for something. When we speak of a template, we say it's a "something template" not a "template something." This may sound like a trivial difference, and it's even a commonly-heard misnomer, but there's actually an important distinction to be made.

Say we're talking about functions. If we say "template function" then we use "template" as an adjective, as though that's what kind of function it is. This is how we properly talk about "static functions" and "member functions" or even "static member functions." Those are all adjectives that clarify what kind of function we're talking about.

This is not the case with templates. We never write a "template function" but instead a "template for a function." This is then shortened to just "function template." The template can be used to make a function which is then usable like any other function. This process is known as "instantiation." That's the same term we use when we make an object of a class type: the object is an instance of the class.

Template instantiation always takes place at compile time. We may pass arguments to the template in order to control how the template is instantiated into a run-time entity. This is conceptually similar to calling a constructor.

C++'s approach with templates differs from C#'s approach with generics. Rather than instantiating templates at compile time, the C# compiler generates MSIL that describes generic types and methods. The runtime then instantiates generics at the point where they're first used. The generic is instantiated for each primitive (`int`, `float`, etc.) and once for all reference types (`string`, `GameObject`, etc.). Run-time

implementations of this differ greatly. For example, IL2CPP instantiates generics at compile time but Microsoft runtimes instantiate them at run time.

With this in mind, let's start looking at the kinds of templates we can make.

Variables

Perhaps the simplest form of template is a template for a variable. Consider the case of defining π:

```
constexpr float PI_FLOAT = 3.14f;
constexpr double PI_DOUBLE = 3.14;
constexpr int32_t PI_INT32 = 3;
```

This requires us to create many variables, each with essentially the same value. We have to come up with unique names for them which adds a lot of noise to our code.

Now let's look at how we'd do this with a variable template:

```
template<typename T>
constexpr T PI = 3.14;
```

First, we start with the `template` keyword and the "template parameters" in angle brackets: `<typename T>`. We'll go in-depth into the various options for template parameters in the next chapter. For now, we'll use the simple `typename T`. This says the template takes one parameter, a "type name" with the name `T`. This is just like parameters to functions. We state the type of the required parameters and what we want to refer to them as in the function: `int i`.

After the `template` we have the thing that the template creates when instantiated. In this case we have a variable named `PI`. It has access to the template parameters, which in this case is just `T`. Here we use `T` as the type of the variable: `T PI`. Just like other variables, we're

free to make it `constexpr` and initialize it: `= 3.14`. We could also make it a pointer, a reference, `const`, or use other forms of initialization like `{3.14}`.

Now that we've defined a template of a variable, let's instantiate it so we have an actual variable:

```
float pi = PI<float>;
DebugLog(pi); // 3.14
```

Instantiation involves naming the template (`PI`) and providing arguments for the required template parameters. This is just like calling a function except that we're using angle brackets (`<>`) instead of parentheses (`()`). We're also passing a type name (`float`) instead of an object (`3.14`).

When the compiler sees this, it looks at the template (`PI`) and matches up our template arguments (`float`) to the template parameters (`T`). It then substitutes the arguments wherever they're used in the templated entity. In this case `T` is replaced with `float`, so we get this:

```
constexpr float PI = 3.14;
```

Then our usage of the template is replaced with usage of the instantiated template, so we get this:

```
float pi = PI;
DebugLog(pi); // 3.14
```

Looking back at the original example where we had `double` and `int32_t` versions of π, we can now replace those with uses of the `PI` template:

```
float pif = PI<float>;
DebugLog(pif); // 3.14

double pid = PI<double>;
DebugLog(pid); // 3.14

int32_t pii = PI<int32_t>;
DebugLog(pii); // 3
```

This example instantiates the `PI` variable template three times. The first instantiation passes `float` as the argument for `T` and the second and third instantiations pass `double` and `int32_t`. This causes the compiler to generate three variables:

```
constexpr float PI = 3.14;
constexpr double PI = 3.14;
constexpr int32_t PI = 3.14;
```

There are two apparent problems here. First, we're initializing an `int32_t` with `= 3.14`. This is fine since `3.14` will be truncated to `3` per the initialization rules. If we didn't want that behavior, we could have used `constexpr T PI{3.14}` and `PI<int32_t>` would then cause a compiler error as `int32_t{3.14}` is not allowed.

Second, we apparently have three variables named `PI` and therefore have a naming conflict. The compiler steps in and generates unique

names. It may call them `PI_FLOAT`, `PI_DOUBLE`, and `PI_INT32` or any other names it deems appropriate so long as they're unique.

This is the same process that occurs when we overload functions: the compiler generates unique names for those functions. When we refer to the function, such as by calling it, the compiler determines which one we're calling and substitutes `Foo()` with `Foo_Void()` or whatever it named the function. With templates, the compiler substitutes `PI<float>` with `PI_FLOAT`.

Lastly, we can explicitly instantiate a variable template without using it for any particular purpose:

```
template constexpr float PI<float>;
```

This is commonly used in libraries that don't need to use the template but rather want to make sure it's compiled into a static or dynamic library so it's available for users at link time.

Functions

Function templates can be instantiated to produce a function just like how variable templates can be instantiated to produce a variable. For example, here's a template for functions that return the maximum of two arguments:

```
// Function template
template<typename T>
T Max(T a, T b)
{
    return a > b ? a : b;
}

// int version
int maxi = Max<int>(2, 4);
DebugLog(maxi); // 4

// float version
float maxf = Max<float>(2.2f, 4.4f);
DebugLog(maxf); // 4.4

// double version
double maxd = Max<double>(2.2, 4.4);
DebugLog(maxd); // 4.4
```

Again we see the `template<typename T>` that begins a template. After it we wrote a function instead of a variable. That function has

access to the type name argument `T`. It uses it as the type of the two parameters as well as the return value.

Then we see three instantiations of the `Max` template: `Max<int>`, `Max<float>`, and `Max<double>`. Just like with variables, the compiler instantiates three functions by substituting the template argument (`int`, `float`, or `double`) anywhere the template parameter `T` is used in the function template:

```
int MaxInt(int a, int b)
{
    return a > b ? a : b;
}

float MaxFloat(float a, float b)
{
    return a > b ? a : b;
}

double MaxDouble(double a, double b)
{
    return a > b ? a : b;
}
```

Then the three function calls that caused this instantiation are replaced by calls to the instantiated functions:

```
// int version
int maxi = MaxInt(2, 4);
```

```
DebugLog(maxi); // 4

// float version
float maxf = MaxFloat(2.2f, 4.4f);
DebugLog(maxf); // 4.4

// double version
double maxd = MaxDouble(2.2, 4.4);
DebugLog(maxd); // 4.4
```

Also, as with any template, we're not limited to just primitive types. We can use *any* type:

```
struct Vector2
{
    float X;
    float Y;

    bool operator>(const Vector2& other) const
    {
        return X > other.X && Y > other.Y;
    }
};

// Vector2 version
Vector2 maxv = Max<Vector2>(Vector2{4, 6}, Vector2{2,
4});
DebugLog(maxv.X, maxv.Y); // 4, 6
```

The implication here is that a template places prerequisites on its parameters. The `Max` template requires that there's a `T > T` operator available. That's definitely satisfied by `int`, `float`, and `double`, but we needed to write an overloaded `Vector2 > Vector2` operator in order for it to work with `Vector2`. Without this operator, we'd get a compiler error:

```
struct Vector2
{
    float X;
    float Y;
};

template <typename T>
T Max(T a, T b)
{
    // Compiler error:
    // "Invalid operands to binary expression (Vector2
and Vector2)"
    return a > b ? a : b;
}

Vector2 maxv = Max<Vector2>(Vector2{4, 6}, Vector2{2,
4});
DebugLog(maxv.X, maxv.Y); // 4, 6
```

Another option, as we've seen before, available to us in C++20 is to implicitly create function templates using `auto` parameters, `auto`

return types, or both. These are known as “abbreviated function templates:”

```
// Abbreviated function template
auto Max(auto a, auto b)
{
    return a > b ? a : b;
}

// Usage is identical
Vector2 maxv = Max<Vector2>(Vector2{4, 6}, Vector2{2,
4});
DebugLog(maxv.X, maxv.Y); // 4, 6
```

Finally, function templates can be explicitly instantiated like this:

```
template bool IsOrthogonal<Vector2>(Vector2, Vector2);
```

Classes

The next kind of template we can create is a template for a `class`, `struct`, or `union`. As with variables and functions, we start with `template<params>` and then write a class:

```
template<typename T>
struct Vector2
{
    T X;
    T Y;

    T Dot(const Vector2<T>& other) const
    {
        return X*other.X + Y*other.Y;
    }
};
```

Notice how the class uses the template argument `T` in place of specific types like `float`. This is allowed anywhere a specific type would otherwise go, such as in the types of data members like `X` and `Y`, the return type of member functions like `Dot`, or in parameters like `other`.

Here's how we'd instantiate this class template to create vectors of a few different types:

```
Vector2<float> v2f{0, 1};
DebugLog(v2f.X, v2f.Y); // 0, 1
```

```
DebugLog(v2f.Dot({1, 0})); // 0

Vector2<double> v2d{0, 1};
DebugLog(v2d.X, v2d.Y); // 0, 1
DebugLog(v2d.Dot({1, 0})); // 0

Vector2<int32_t> v2i{0, 1};
DebugLog(v2i.X, v2i.Y); // 0, 1
DebugLog(v2i.Dot({1, 0})); // 0
```

The compiler-instantiated `Vector2` classes then look like this:

```
struct Vector2Float
{
    float X;
    float Y;

    float Dot(const Vector2Float& other) const
    {
        return X*other.X + Y*other.Y;
    }
};

struct Vector2Double
{
    double X;
    double Y;
```

```
    double Dot(const Vector2Double& other) const
    {
        return X*other.X + Y*other.Y;
    }
};

struct Vector2Int32
{
    int32_t X;
    int32_t Y;

    int32_t Dot(const Vector2Int32& other) const
    {
        return X*other.X + Y*other.Y;
    }
};
```

Then the usages of the class template are replaced with usages of these instantiated classes:

```
Vector2Float v2f{0, 1};
DebugLog(v2f.X, v2f.Y); // 0, 1
DebugLog(v2f.Dot({1, 0})); // 0

Vector2Double v2d{0, 1};
DebugLog(v2d.X, v2d.Y); // 0, 1
DebugLog(v2d.Dot({1, 0})); // 0
```

```
Vector2Int32 v2i{0, 1};
DebugLog(v2i.X, v2i.Y); // 0, 1
DebugLog(v2i.Dot({1, 0})); // 0
```

Note that the `Dot` calls are particularly compact and valid only because of several rules we've seen so far. Take the case of `v2f` which is a `Vector2<float>`. When we call `v2f.Dot({1, 0})`, the compiler looks at the `Dot` it instantiated as part of the `Vector2` template. That `Dot` takes a `const Vector2<float>&` parameter, so the compiler interprets `{1, 0}` as aggregate initialization of a `Vector2<float>`. Because `{0, 1}` doesn't have a name, that `Vector2<float>` is an rvalue. It can be passed to a `const` lvalue reference and it's lifetime is extended until after `Dot` returns.

Class templates can be explicitly instantiated like this:

```
template struct Vector2<float>;
template struct Vector2<double>;
template struct Vector2<int32_t>;
```

Members

Classes can include templates for member functions:

```
struct Vector2
{
    float X;
    float Y;

    template<typename T>
    bool IsNearlyZero(T threshold) const
    {
        return X < threshold && Y < threshold;
    }
};
```

Even though `Vector2` isn't a template, it can contain a member function template. We use it just like a normal member function:

```
Vector2 vec{0.5f, 0.5f};

// Float
DebugLog(vec.IsNearlyZero(0.6f)); // true

// Double
DebugLog(vec.IsNearlyZero(0.1)); // false
```

```
// Int
DebugLog(vec.IsNearlyZero(1)); // true
```

This causes the compiler to instantiate `IsNearlyZero` three times:

```
struct Vector2
{
    float X;
    float Y;

    bool IsNearlyZeroFloat(float threshold) const
    {
        return X < threshold && Y < threshold;
    }

    bool IsNearlyZeroDouble(double threshold) const
    {
        return X < threshold && Y < threshold;
    }

    bool IsNearlyZeroInt(int threshold) const
    {
        return X < threshold && Y < threshold;
    }
};
```

Then the calls to the member function template are replaced with calls to the instantiated functions:

```
Vector2 vec{0.5f, 0.5f};

// Float
DebugLog(vec.IsNearlyZeroFloat(0.6f)); // true

// Double
DebugLog(vec.IsNearlyZeroDouble(0.1)); // false

// Int
DebugLog(vec.IsNearlyZeroInt(1)); // true
```

We can also write templates for `static` member variables:

```
struct HealthRange
{
    template<typename T>
    constexpr static T Min = 0;

    template<typename T>
    constexpr static T Max = 100;
};
```

They're used the same way other `static` member variables are:

```
float min = HealthRange::Min<float>;
int32_t max = HealthRange::Max<int32_t>;
DebugLog(min, max); // 0, 100
```

As expected, the compiler instantiates these templates like so:

```
struct HealthRange
{
    constexpr static float MinFloat = 0;
    constexpr static int32_t MaxInt = 100;
};
```

And then replaces the member variable template usage with these instantiated member variables:

```
float min = HealthRange::MinFloat;
int32_t max = HealthRange::MaxInt;
DebugLog(min, max); // 0, 100
```

Lastly, we can write templates for member classes:

```
struct Math
{
    template<typename T>
    struct Vector2
    {
        T X;
        T Y;
    };
};
```

These can then be used like normal member classes:

```
Math::Vector2<float> v2f{2, 4};
DebugLog(v2f.X, v2f.Y); // 2, 4

Math::Vector2<double> v2d{2, 4};
DebugLog(v2d.X, v2d.Y); // 2, 4
```

The compiler then performs the usual instantiation and replacement:

```
struct Math
{
    struct Vector2Float
    {
        float X;
        float Y;
    };

    struct Vector2Double
    {
        double X;
        double Y;
    };
};

Math::Vector2Float v2f{2, 4};
DebugLog(v2f.X, v2f.Y); // 2, 4
```

```
Math::Vector2Double v2d{2, 4};
DebugLog(v2d.X, v2d.Y); // 2, 4
```

Finally, explicit instantiation of member templates looks like this:

```
// Member function template
template bool Vector2::IsNearlyZero<float>(float) const;

// Member variable template
template const float HealthRange::Min<float>;

// Member class template
template struct Math::Vector2<float>;
```

Lambdas

Since lambdas are compiler-generated classes, their overloaded `operator()` can also be templated. Prior to C++20, the “abbreviated function template” syntax based on `auto` parameters and return values needed to be used:

```
// "Abbreviated function template" of LambdaClass' operator()
auto madd = [](auto x, auto y, auto z) { return x*y + z; };

// Instantiate with float
DebugLog(madd(2.0f, 3.0f, 4.0f)); // 10

// Instantiate with int
DebugLog(madd(2, 3, 4)); // 10
```

The compiler-generated lambda class will then look something like this:

```
struct Madd
{
    // Abbreviated function template
    auto operator()(auto x, auto y, auto z) const
    {
        return x*y + z;
```

```
    }
};
```

Then the two calls to its `operator()` cause the abbreviated function template to be instantiated into an overload set:

```
struct Madd
{
    float operator()(float x, float y, float z) const
    {
        return x*y + z;
    }

    int operator()(int x, int y, int z) const
    {
        return x*y + z;
    }
};
```

The compiler will then replace the lambda syntax with instantiation of these classes and calls to their `operator()`:

```
Madd madd{};

// Call (float, float, float) overload of operator()
DebugLog(madd(2.0f, 3.0f, 4.0f)); // 10
```

```
// Call (int, int, int) overload of operator()
DebugLog(madd(2, 3, 4)); // 10
```

Starting in C++20, we can use the normal, non-abbreviated, style of template syntax. The compiler generates exactly the same code with this version as it did with the “abbreviated” syntax:

```
// Lambda with explicit template and "trailing return
type"
auto madd = []<typename T>(T x, T y, T z) -> T { return
x*y + z; };

// Instantiate with float
DebugLog(madd(2.0f, 3.0f, 4.0f)); // 10

// Instantiate with int
DebugLog(madd(2, 3, 4)); // 10
```

Lambdas’ templated `operator()` can’t be explicitly instantiated.

C# Equivalency

As we've seen, C++ has quite a different take on generic programming than C#. Its templates can be applied not only to classes, structs, and member functions, but also lambdas, variables, functions, member variables, and member functions. All of these except lambdas can be emulated in C# by wrapping them in a `static` class:

```
// C#
public static class Wrapper<T>
{
    // Variable
    public static readonly T Default = default(T);

    // Function
    public static string SafeToString(T obj)
    {
        return object.ReferenceEquals(obj, null) ? "" :
obj.ToString();
    }
}

// Variable
DebugLog(Wrapper<float>.Default); // 0

// Function
DebugLog(Wrapper<Player>.SafeToString(new Player())); //
```

```
"Player 1"
DebugLog(Wrapper<Player>.SafeToString(null)); // ""
```

The major difference comes in when we consider what we're allowed to do with type parameters. By default, a C# type parameter is treated like a `System.Object`/`object`. That means it has almost no functionality beyond basics like the `ToString()` and `default(T)` expressions we used above.

We can add restrictions using `where` constraints to enable more functionality, but this is very limited. There are a handful of constraints like `where T : new()` that allow us to call a default constructor or `where T : class` that allow us to use `null`, but mostly we use `where T : ISomeInterface` or `where T : SomeBaseClass` to enable calls to `virtual` functions in the interface or base class.

Let's try porting one of the above `Vector2` examples from C++ to C#:

```
template<typename T>
struct Vector2
{
    T X;
    T Y;

    T Dot(const Vector2<T>& other) const
    {
        return X*other.X + Y*other.Y;
    }
};
```

First, a literal translation of the syntax looks like this:

```
// C#
public struct Vector2<T>
{
    public T X;
    public T Y;

    public T Dot(in Vector2<T> other)
    {
        return X*other.X + Y*other.Y;
    }
}
```

Other than needing to make `Dot` non-`const`, because that's not supported in C#, not much has changed. The problem is that we now get compiler errors on `X*other.X`, `Y*other.Y`, and the `+` operator between them:

> Operator '*' cannot be applied to operands of type 'T' and 'T'*
> *Operator '*' cannot be applied to operands of type 'T' and 'T'*
> Operator '+' cannot be applied to operands of type 'T' and 'T'

The compiler is treating `T` as an `object` and both `object*object` and `object+object` are compiler errors. It doesn't matter that we only ever use types like `float` and `double` that *do* support the `*` and `+` operators. The compiler insists that *all* possible types that could be used for `T` support `T*T` and `T+T`. Since that's not the case for types like `Player`, we get compiler errors.

So we'll need to add a `where` constraint in order to restrict `T` to a subset of types that does support `*` and `+`. Looking over the list of

options, we don't see anything like a `where T : T*T` or `where T : T+T`. Our only option is to avoid `*` and `+` and instead call `virtual` functions named `Multiply` and `Add` in an implemented interface or base class because we can write `where` constraints for those.

Here's such an interface:

```
// C#
public interface IArithmetic<T>
{
    T Multiply(T a, T b);
    T Add(T a, T b);
}
```

Here's a `struct` that implements it for `float`:

```
// C#
public struct FloatArithmetic : IArithmetic<float>
{
    public float Multiply(float a, float b)
    {
        return a * b;
    }

    public float Add(float a, float b)
    {
        return a + b;
    }
}
```

Now we can pass a `FloatArithmetic` to `Vector2` for it to call `IArithmetic.Multiply` and `IArithmetic.Add` on instead of the built-in `*` and `+` operators:

```
// C#
public struct Vector2<T, TArithmetic>
    where TArithmetic : IArithmetic<T>
{
    public T X;
    public T Y;
    private TArithmetic Arithmetic;

    public Vector2(T x, T y, TArithmetic arithmetic)
    {
        X = x;
        Y = y;
        Arithmetic = arithmetic;
    }

    public T Dot(Vector2<T, TArithmetic> other)
    {
        T xProduct = Arithmetic.Multiply(X, other.X);
        T yProduct = Arithmetic.Multiply(Y, other.Y);
        return Arithmetic.Add(xProduct, yProduct);
    }
}
```

Here's how we'd use this:

```
// C#
var vecA = new Vector2<float, FloatArithmetic>(1, 0,
default);
var vecB = new Vector2<float, FloatArithmetic>(0, 1,
default);
DebugLog(vecA.Dot(vecB)); // 0
```

While this design works, it's created several problems. First, we have a lot of boilerplate in `IArithmetic`, `FloatArithmetic`, extra type arguments to the generics (`<T, TArithmetic>` instead of just `<T>`), and an extra `arithmetic` parameter to the constructor. That's a hit to productivity and readability, but at least not a concern that translates much to the executable the compiler generates.

The second issue is that our `Vector2` has increased in size since it includes an `Arithmetic` field. That's a managed reference to an `IArithmetic`. On a 64-bit CPU, it'll take up at least 8 bytes. Since `X` and `Y` both take up 4 bytes, the size of `Vector2` has doubled. This will impact memory usage and, perhaps more importantly, cache utilization as only half as many vectors can now fit in a cache line.

The third issue is that `FloatArithmetic` needs to be "boxed" from a `struct` into a managed `IArithmetic` reference. This will create garbage for the garbage collector to later collect. In the above example, this happens with each call to the `Vector2` constructor. This deferred performance cost may cause frame hitches or other issues.

To avoid the boxing, we could switch from a `struct` to a `class` and share a global instance::

```
// C#
public class FloatArithmetic : IArithmetic<float>
{
    public static readonly FloatArithmetic Default = new
FloatArithmetic();

    public float Multiply(float a, float b)
    {
        return a * b;
    }

    public float Add(float a, float b)
    {
        return a + b;
    }
}

var vecA = new Vector2<float, FloatArithmetic>(1, 0,
FloatArithmetic.Default);
var vecB = new Vector2<float, FloatArithmetic>(0, 1,
FloatArithmetic.Default);
DebugLog(vecA.Dot(vecB)); // 0
```

This presents another performance issue: our reads of `FloatArithmetic.Default` may read from “cold” memory, i.e. memory that’s not in a CPU cache.

The fourth and final issue is that the calls to `Arithmetic.Multiply` and `Arithmetic.Add` in `Dot` are `virtual` function calls because all

functions in interfaces are implicitly `virtual`. In some cases, the compiler will be able to conclusively determine that `TArithmetic` is `FloatArithmetic` and "de-virtualize" the calls. In many other cases, we'll suffer the runtime overhead of three virtual function calls per `Dot`.

Another approach is to wrap the `float` values in a class that implements an interface with `Multiply` and `Add`:

```
// C#
public interface INumeric<T>
{
    INumeric<T> Create(T val);
    T Value { get; set; }
    INumeric<T> Multiply(T val);
    INumeric<T> Add(T val);
}

public class FloatNumeric : INumeric<float>
{
    public float Value { get; set; }

    public FloatNumeric(float val)
    {
        Value = val;
    }

    public INumeric<float> Create(float val)
    {
        return new FloatNumeric(val);
```

```
    }

    public INumeric<float> Multiply(float val)
    {
        return Create(Value * val);
    }

    public INumeric<float> Add(float val)
    {
        return Create(Value + val);
    }
}
```

`Vector2` can now hold `INumeric` fields and call its `virtual` functions:

```
// C#
public struct Vector2<T, TNumeric>
    where TNumeric : INumeric<T>
{
    public TNumeric X;
    public TNumeric Y;

    public Vector2(TNumeric x, TNumeric y)
    {
        X = x;
        Y = y;
    }
```

```
    public T Dot(Vector2<T, TNumeric> other)
    {
        INumeric<T> xProduct = X.Multiply(other.X.Value);
        INumeric<T> yProduct = Y.Multiply(other.Y.Value);
        INumeric<T> sum = xProduct.Add(yProduct.Value);
        return sum.Value;
    }
}

var vecA = new Vector2<float, FloatNumeric>(
    new FloatNumeric(1),
    new FloatNumeric(0)
);
var vecB = new Vector2<float, FloatNumeric>(
    new FloatNumeric(0),
    new FloatNumeric(1)
);
DebugLog(vecA.Dot(vecB)); // 0
```

This suffers the same problems as the previous approach: garbage creation for each `FloatNumeric` that's created, virtual function calls to `Add` and `Multiply`, extraneous type parameters, boilerplate, etc. At this point, many C# programmers will give up and manually "instantiate" the `Vector2` "template" like a C++ compiler would:

```
// C#
public struct Vector2Float
{
    public float X;
```

```
    public float Y;

    public float Dot(in Vector2Float other)
    {
        return X*other.X + Y*other.Y;
    }
}

public struct Vector2Double
{
    public double X;
    public double Y;

    public double Dot(in Vector2Double other)
    {
        return X*other.X + Y*other.Y;
    }
}

public struct Vector2Int
{
    public int X;
    public int Y;

    public int Dot(in Vector2Int other)
    {
        return X*other.X + Y*other.Y;
    }
```

```
}

var vecA = new Vector2Float{X=1, Y=0};
var vecB = new Vector2Float{X=0, Y=1};
DebugLog(vecA.Dot(vecB)); // 0
```

This is efficient, but now suffers all the usual issues with code duplication: the need to change many copies, bugs when the copies get out of sync, etc. To address this, we may turn to a code generation tool that uses some form of templates to generates `.cs` files. This may be run in an earlier build step, but it won't be integrated into the main codebase, may require additional languages, still requires unique naming, and a variety of other issues.

C++ avoids the code duplication, the external tools, the virtual function calls, the cold memory reads, the boxing and garbage collection, the need for an interface and boilerplate implementation of it, the extra type parameters, and the `where` constraints. Instead, it simply produces a compiler error when `T*T` or `T+T` is a syntax error.

Conclusion

We've barely scratched the surface of templates and already we've seen that they're far more powerful than C# generics. We can easily write code like `Dot` that's simultaneously efficient, readable, and generic. C# struggles with even simple examples like this and often requires us to sacrifice one or more of these qualities.

26. Template Parameters

Type Template Parameters

All of the examples of templates in the intro chapter took one parameter:

```
template<typename T>
```

That's often enough to create many templates as we've seen from types like C#'s `List<T>` and `NativeArray<T>` generic classes. Still, many others like `Dictionary<TKey, TValue>` require more parameters. Adding these is simple and just like adding function parameters:

```
// Class template that takes two parameters
template<typename TKey, typename TValue>
struct Pair
{
    TKey Key;
    TValue Value;
};

// Specify both parameters to instantiate the template
Pair<int, float> pair{123, 3.14f};

DebugLog(pair.Key, pair.Value); // 123, 3.14
```

Also similar to function parameters, but unlike type parameters to C# generics, we can specify default values:

```
// Second parameter has a default value
template<typename TKey, typename TValue=int>
struct Pair
{
    TKey Key;
    TValue Value;
};

// Only need to pass one parameter
// The second gets the default: int
Pair<int> pair1{123, 456};
DebugLog("TValue is int?", typeid(pair1.Value) ==
typeid(int)); // true
DebugLog(pair1.Key, pair1.Value); // 123, 456

// We can still pass two parameters
Pair<int, float> pair2{123, 3.14f};
DebugLog("TValue is int?", typeid(pair2.Value) ==
typeid(int)); // false
DebugLog(pair2.Key, pair2.Value); // 123, 3.14
```

It's also common to see `class` instead of `template` in the template parameters list. There is no difference between the two. Non-class types like `int` and `float` are perfectly usable with `class` template parameters. The choice of which to use is mostly one of style:

```
// Parameters are "class" instead of "typename"
// Behaves the same
template<class TKey, class TValue=int>
struct Pair
{
    TKey Key;
    TValue Value;
};

// Non-class types are still usable as template arguments
Pair<int> pair1{123, 456};
Pair<int, float> pair2{123, 3.14f};
```

Unlike C#, the names of the parameters are optional, even when they have default values:

```
// Parameter names omitted, with and without default values
template<class, class=int>
void DoNothing()
{
}

DoNothing<float>();
DoNothing<float, int>();
```

The types also don't need to be defined, only declared, in order to be used to instantiate a template. This works as long as the template doesn't use the type, similar to if it wasn't given a name at all:

```
// Declared, but not defined
struct Vector2;

// Template that doesn't use its type parameter
template<typename T>
void DoNothing()
{
}

// OK because Vector2 doesn't actually get used
DoNothing<Vector2>();
```

In the case that a template parameter has the same name as a name outside of the template, there's no collision as the context makes it clear which one is being referred to:

```
struct Vector
{
    float X = 0;
    float Y = 0;
};

// Template parameter has the same name as a class
outside the template: Vector
template<typename Vector>
```

```
Vector Make()
{
    return Vector{};
}

// "int" used as the type named "Vector"
auto val = Make<int>();
DebugLog(val); // 0

// "Vector" doesn't refer to the type parameter
// The template isn't referenced here
auto vec = Vector{};
DebugLog(vec.X, vec.Y); // 0, 0
```

Template Template Parameters

Consider a `Map` class template that holds keys and values via a `List<T>` class template:

```
template<typename T>
struct List
{
    // ... implementation similar to C#
};

template<typename TKey, typename TValue>
struct Map
{
    List<TKey> Keys;
    List<TValue> Values;
};

Map<int, float> map;
```

The `Map` template always uses the `List` template. If we wanted to abstract the kind of container that holds the keys and values to make `Map` more flexible, we could use a “template template parameter.” This is where we pass a template like `List`, not an instantiation of a template like `List<T>`, as a parameter to a template:

```
template<typename T>
struct List
```

```
{
    // ... implementation similar to C#
};

template<typename T>
struct FixedList
{
    // ... implementation similar to C# except that it's
a fixed size
};

// The third parameter is a template, not a type
// That template needs to take one type parameter
template<typename TKey, typename TValue,
template<typename> typename TContainer>
struct Map
{
    // Use the template parameter instead of directly
using List
    TContainer<TKey> Keys;
    TContainer<TValue> Values;
};

// Pass List, which is a template taking one type
parameter, as the parameter
// Do not pass an instantiation of the template like
List<int>
Map<int, float, List> listMap;
```

```
// Pass FixedList as the parameter
// It also takes one type parameter
Map<int, float, FixedList> fixedListMap;
```

When we do this, the compiler instantiates these two classes for `listMap` and `fixedListMap`:

```
struct MapList
{
    List<int> Keys;
    List<float> Values;
};

struct MapFixedList
{
    FixedList<int> Keys;
    FixedList<float> Values;
};
```

Template template parameters can also have default values:

```
template<
    typename TKey,
    typename TValue,
    template<typename> typename TKeysContainer=List,
    template<typename> typename TValuesContainer=List>
struct Map
```

```
{
    TKeysContainer<TKey> Keys;
    TValuesContainer<TValue> Values;
};

// TKeysContainer=List, TValuesContainer=List
Map<int, float> map1;

// TKeysContainer=FixedList, TValuesContainer=List
Map<int, float, FixedList> map2;

// TKeysContainer=FixedList, TValuesContainer=FixedList
Map<int, float, FixedList, FixedList> map3;
```

Non-Type Template Parameters

The third kind of template parameter is known as a “non-type template parameter.” These are compile-time constant values, not the names of types or templates. For example, we can use this to write the `FixedList` type backed by an array data member:

```
// Size is a "non-type template parameter"
// A compile-time constant needs to be passed
template<typename T, int Size>
struct FixedList
{
    // Use Size like any other int
    T Elements[Size];

    T& operator[](int index)
    {
        return Elements[index];
    }

    int GetLength() const noexcept
    {
        return Size;
    }
};

// Pass 3 for Size
FixedList<int, 3> list1;
```

```
list1[1] = 123;
DebugLog(list1[1]); // 123

// Pass 2 for Size
FixedList<float, 2> list2;
list2[0] = 3.14f;
DebugLog(list2[0]); // 3.14
```

Just like with “type template parameters” and “template template parameters,” the compiler substitutes the value anywhere it’s used when instantiating the template:

```
struct FixedListInt3
{
    int Elements[3];

    int& operator[](int index)
    {
        return Elements[index];
    }

    int GetLength() const noexcept
    {
        return 3;
    }
};

struct FixedListFloat2
```

```
{
    float Elements[2];

    float& operator[](int index)
    {
        return Elements[index];
    }

    int GetLength() const noexcept
    {
        return 2;
    }
};
```

Default values are allowed for non-type template parameters, too:

```
// Template parameters control the initial capacity and
growth factor
template<typename T, int InitialCapacity=4, int
GrowthFactor=2>
class List
{
    // ... implementation
};
```

We can now use these to tune the performance of our `List` classes based on expected usage:

```
// Defaults are acceptable
List<int> list1;

// Start off with a lot of capacity
List<int, 1024> list2;

// Don't start with a little capacity, but grow fast
List<int, 4, 10> list3;

// Start empty and grow by doubling
List<int, 0, 2> list4;
```

The kinds of values we can pass to non-type template parameters is restricted to "structural types." The first such kind of "structural type" is the one we've already seen: integers. We can use any size (`short`, `long`, etc.) and it doesn't matter if it's signed or not (`signed`, `unsigned`, no specifier). This also includes quasi-integers like `char` and the type of `nullptr`.

The second kind of values are pointers and lvalue references:

```
// Takes a pointer and a reference to some type T
template<typename T, const T* P, const T& R>
constexpr T Sum = *P + R;

// A constant array and a constant integer
constexpr int a[] = { 100 };
constexpr int b = 23;
```

```
// The 'a' array "decays" to a pointer
// The 'b' integer is an lvalue because it has a name: b
constexpr int sum = Sum<int, a, b>;

DebugLog(sum); // 123
```

The third kind is similar: pointers to members.

```
// A class with two int data members
struct Player
{
    int Health = 100;
    int Armor = 50;
};

// Template for a function that gets an int data member
// Takes the type of the class and a pointer to one of
its int data members
template<typename TCombatant, int TCombatant::* pStat>
constexpr int GetStat(const TCombatant& combatant)
{
    return combatant.*pStat;
}

// Get both int data members via the function template
and pointers to members
Player player;
DebugLog(GetStat<Player, &Player::Health>(player)); //
```

```
100
DebugLog(GetStat<Player, &Player::Armor>(player)); // 50
```

Starting in C++20, there are two more kinds. First, floating point types like `float` and `double`:

```
template<float MinValue, float MaxValue>
float Clamp(float value)
{
    return x > MaxValue ? MaxValue : x < MinValue ?
MinValue : value;
}

DebugLog(Clamp<0, 100>(50)); // 50
DebugLog(Clamp<0, 100>(150)); // 100
DebugLog(Clamp<0, 100>(-50)); // 0
```

Second, “literal types” we’ve seen before when writing compile-time code:

```
// As a simple aggregate, this is a "literal type"
struct Pixel
{
    int X;
    int Y;
};

// Template taking a "literal type"
```

```
template<Pixel P>
bool IsTopLeft()
{
    return P.X == 0 && P.Y == 0;
}

// Passing a "literal type" as a template argument
DebugLog(IsTopLeft<Pixel{2, 4}>()); // false
DebugLog(IsTopLeft<Pixel{0, 0}>()); // true
```

Regardless of language version, there are some additional restrictions on the kinds of expressions we can pass as a template argument. First, we can't pass pointers or references to sub-objects such as base classes and array elements:

```
template<const int& X>
constexpr int ValOfTemplateParam = X;

constexpr int a[] = { 100 };

// Compiler error: can't reference sub-object of a as
non-type template param
constexpr int val = ValOfTemplateParam<a[0]>;
```

Temporary objects also can't be passed:

```
// Compiler error: can't pass a temporary object
constexpr int val = ValOfTemplateParam<123>;
```

Nor can string literals:

```
template<const char* str>
void Print()
{
    DebugLog("Letter:", *str);
}

constexpr char c = 'A';

// Compiler error: can't pass a string literal
Print<"hi">();

Print<&c>(); // Letter: A
```

And finally, whatever type `typeid` returns can't be passed as a template argument:

```
template<decltype(typeid(char)) tid>
void PrintTypeName()
{
    DebugLog(tid.name());
}

// Compiler error: can't pass what typeid evaluates to
PrintTypeName<typeid(char)>();
```

While they’re not strictly prohibited, it’s important to know that arrays in template parameters are implicitly converted to pointers. This can have some important consequences:

```
// Array parameter is automatically transformed to a
pointer
template<const int X[]>
constexpr void PrintSizeOfArray()
{
    // Bug: prints the size of a pointer, not the size of
the array
    DebugLog(sizeof(X));
}

constexpr int32_t arr[3] = { 100, 200, 300 };

// Bug
PrintSizeOfArray<arr>(); // 8 (on 64-bit CPUs)

// OK
DebugLog(sizeof(arr)); // 12
```

Ambiguity

There are a few cases that can arise where template parameters appear ambiguous. Similar to operator precedence, there are clear rules that determine how the compiler disambiguates template parameters. These cases usually don't arise as programmers make good choices with naming, but it's important to know the rules to be able to figure out what the compiler is doing in edge cases.

The first case happens when a member template is declared outside the class with a parameter that has the same name as a member of the class it's a member of. In this case, the member of the class is used instead of the template parameter:

```
// Class template with one type parameter: T
template<class T>
struct MyClass
{
    // Member class
    struct Thing
    {
    };

    // Member function declaration, not definition
    int GetSizeOfThing();
};

// Member function definition outside the class
// Uses 'Thing' instead of 'T' as the class' type
parameter name
```

```
// 'Thing' is the same name as the member class 'Thing'
template<class Thing>
int MyClass<Thing>::GetSizeOfThing()
{
    // 'Thing' refers to the member class, not the type
parameter
    return sizeof(Thing);
}

// Instantiate the class template with T=double
MyClass<double> mc{};

// Call the member function on a MyClass<double>
// Returns the size of the member class: 1 for an empty
struct
DebugLog(mc.GetSizeOfThing()); // 1, not 8
```

The second case also happens when a member of a class template is defined outside the class template. Specifically, it only happens when the name of a parameter is the same as the name of a member of the namespace the class is in. In this case, we get the opposite: the type parameter is used instead of the namespace member.

```
namespace MyNamespace
{
    // Class member of the namespace
    class Thing
    {
```

```
    };

    // Class template with one type parameter: T
    template<class T>
    struct MyClass
    {
        // Member function declaration, not definition
        int GetSizeOfThing(T thing);
    };
}

// Member function definition outside the class
// Uses 'Thing' instead of 'T' as the class' type parameter name
// 'Thing' is the same name as the namespace member class 'Thing'
// 'Thing' is used as the type of a parameter to the function
template<class Thing>
int MyNamespace::MyClass<Thing>::GetSizeOfThing(Thing thing)
{
    // 'Thing' refers to the type parameter, not the namespace member
    return sizeof(Thing);
}

// Instantiate the class template with T=double
```

```
MyNamespace::MyClass<double> mc{};

// Call the member function on a MyClass<double>
// Returns the size of the type parameter: 8 for double
DebugLog(mc.GetSizeOfThing({})); // 8, not 1
```

The third case is when a class template's parameter has the same name as a member of one of its base classes. In this case, the ambiguity goes to the base class' member:

```
struct BaseClass
{
    struct Thing
    {
    };
};

// Class template with one type parameter: Thing
// 'Thing' is the same name as the base class' member class 'Thing'
template<class Thing>
struct DerivedClass : BaseClass
{
    // 'Thing' refers to the base class' member class, not the type parameter
    int Size = sizeof(Thing);
};
```

```
// Instantiate the class template with Thing=double
DerivedClass<double> dc;

// See how big 'Thing' was when initializing 'Size'
// It's the size of BaseClass::Thing: 1 for an empty struct
DebugLog(dc.Size); // 1, not 8
```

Unlike the first two cases, this case is possible in C# as well. Unlike C++, `Thing` refers to the type parameter, not the base class member:

```
// C#
public class BaseClass
{
    public struct Thing
    {
    };
};

// Generic class with one type parameter: Thing
// 'Thing' is the same name as the base class' member class 'Thing'
public class DerivedClass<Thing> : BaseClass
{
    // 'Thing' refers to the type parameter, not base class' member class
    public Type ThingType = typeof(Thing);
};
```

```
// Instantiate the generic class with Thing=double
DerivedClass<double> dc = new DerivedClass<double>();

// See what type 'Thing' was when initializing
'ThingType'
// It's the type parameter 'double', not BaseClass.Thing
DebugLog(dc.ThingType); // System.Double
```

Conclusion

C# generics provide support for type parameters, but not the non-type parameters and template parameters that C++ templates provide support for. Even so, C++ type parameters include additional functionality such as support for default arguments and omitting the name of the parameter.

Template parameters allow for more generic code by using a template like `Container` as a variable rather than a specific template like `List<T>`. Non-type template parameters allow passing compile-time constant expressions so we can use values, not types, in our templates. This allows us to create class templates like `FixedList<T>` with static sizes to avoid dynamic allocation and the cost of dynamic resizing when we don't need it or to tune the allocation strategy of a `List<T>` when we do need dynamic resizing.

27. Template Deduction and Specialization

Template Argument Deduction

The compiler has to know all the arguments to instantiate a template, but that doesn't mean we have to explicitly state them all. Just like how we can use `auto` variables, parameters, and return values and the compiler will deduce their types, the compiler can also deduce template arguments.

The same is true to some extent with C# generics. Consider this example:

```
// C#
static class TypeUtils
{
    // Generic method
    public static void PrintType<T>(T x)
    {
        DebugLog(typeof(T));
    }
}

// Type arguments explicitly specified
TypeUtils.PrintType<int>(123); // System.Int32
TypeUtils.PrintType<bool>(true); // System.Boolean

// Type arguments deduced by the compiler
TypeUtils.PrintType(123); // System.Int32
TypeUtils.PrintType(true); // System.Boolean
```

The same works in C++, as we see in this literal translation of the C#:

```
struct TypeUtils final
{
    // Member function template
    template<typename T>
    static void PrintType(T x)
    {
        DebugLog(typeid(T).name());
    }
};

// Type arguments explicitly specified
TypeUtils::PrintType<int>(123); // i
TypeUtils::PrintType<bool>(true); // b

// Type arguments deduced by the compiler
TypeUtils::PrintType(123); // i
TypeUtils::PrintType(true); // b
```

Support for deduction in C++ is considerably more advanced than in C#. For example, non-type template parameters can be deduced:

```
// Template has one type parameter (T) and one non-type
parameter (N)
template<class T, int N>
```

```
// Function takes a reference to an array of length N
const T elements
int GetLengthOfArray(const T (&t)[N])
{
    return N;
}

// Compiler deduces T as int and N as 3
DebugLog(GetLengthOfArray({1, 2, 3})); // 3

// Compiler deduces T as float and N as 2
DebugLog(GetLengthOfArray({2.2f, 3.14f})); // 2
```

Template template parameters can be deduced, too:

```
// Template with two parameters:
// 1) T, a type parameter
// 2) TContainer, a template parameter
template<typename T, template<typename> typename
TContainer>
void PrintLength(const TContainer<T>& container)
{
    DebugLog(container.Length);
}

template<typename T>
struct List
{
```

```
    int Length;
};

List<int> list{};
PrintLength(list); // T deduced as int, TContainer
deduced as List
```

The compiler will also consider all the overloaded functions in an attempt to find the one that matches best:

```
// Template takes one type parameter
template<class T>
// Function takes a pointer to a function that takes a T
and returns a T
int CallWithDefaultAndReturn(T(*func)(T))
{
    return func({});
}

int AddOne(int x)
{
    DebugLog("int");
    return x + 1;
}

int AddOne(char x)
{
    DebugLog("char");
```

```
    return x + 1;
}

// CallWithDefaultAndReturn is an overload set
// Compiler looks at this function and deduces that T is
int:
//   int AddOne(int)
// Compiler looks at this function and fails to deduce T:
//   int AddOne(char)
// Since deduction succeeded for one of them, that one
gets passed
DebugLog(CallWithDefaultAndReturn(AddOne)); // "int" then
1
```

Note that deduction involves a few transformations of types. First, arrays “decay” to pointers:

```
template<class T>
void ArrayOrPointer(T)
{
    DebugLog("is array?", typeid(T) == typeid(int[3]));
    DebugLog("is pointer?", typeid(T) == typeid(int*));
}

int arr[3];
ArrayOrPointer(arr); // is array? false, is pointer? true
```

Second, functions “decay” to function pointers:

```
void SomeFunction(int) {}

template<class T>
void FunctionOrPointer(T)
{
    DebugLog("is function?", typeid(T) ==
typeid(decltype(SomeFunction)));
    DebugLog("is pointer?", typeid(T) == typeid(void(*)
(int)));
}

FunctionOrPointer(SomeFunction); // is function? false,
is pointer? true
```

And third, `const` is removed:

```
template<class T>
void ConstOrNonConst(T x)
{
    // If T was 'const int' then this would be a compiler
error
    x = {};
}

const int c = 123;
ConstOrNonConst(c); // Compiles, meaning T is non-const
int
```

Fourth, references to `T` become just `T`:

```
template<class T>
void RefDetector(T x)
{
    // If T is a reference, this assigns to the caller's
value
    // If T is not a reference, this assigns to the local
copy
    x = 123;
}

int i = 42;
int& ri = i;
RefDetector(ri);
DebugLog(i); // 42
```

To keep the reference, we have to say that we want a reference by adding the `&`:

```
template<class T>
void RefDetector(T& x) // <-- Added &
{
    x = 123;
}

int i = 42;
```

```
int& ri = i;
RefDetector(ri);
DebugLog(i); // 123
```

One exception is when passing an lvalue to a function template that takes a non-const rvalue reference. In this case, the compiler will deduce the type as an rvalue reference:

```
template<class T>
void Foo(T&&)
{
}

int i = 123; // lvalue, not lvalue reference
Foo(i); // T is int&&
Foo(123); // T is int&
```

After these transformations, the compiler looks for an exact match but it'll also accept a few discrepancies. First, non-`const` will match `const` but not the other way around:

```
template<typename T>
void TakeConstRef(const T& x)
{
}

template<typename T>
void TakeNonConstRef(T& x)
```

```
{
    x = 42;
}

// Compiler deduces T='const int&' even though 'i1' is
non-const
int i1 = 123;
TakeConstRef(i1);

// Compiler deduces T='const int&'
const int i2 = 123;
TakeNonConstRef(i2); // Compiler error: can't assign to x
```

Second, the same is true for pointers:

```
template<typename T>
void TakeConstRef(const T* p)
{
}

template<typename T>
void TakeNonConstRef(T* p)
{
    *p = 42;
}

// Compiler deduces T='const int*' even though 'i1' is
non-const
```

```
int i1 = 123;
TakeConstRef(&i1);

// Compiler deduces T='const int*'
const int i2 = 123;
TakeNonConstRef(&i2); // Compiler error: can't assign to
*p
```

And third, derivation is allowed to support polymorphism:

```
template<class T>
struct Base
{
};

template<class T>
struct Derived : public Base<T>
{
};

template<class T>
void TakeBaseRef(Base<T>&)
{
}

Derived<int> derived;

// Compiler accepts Derived<T> for Base<T> an deduces
```

```
that T is 'int'
TakeBaseRef(derived);
```

Class Template Argument Deduction

Since C++17, the arguments to a class template can also be deduced:

```
// Class template
template<class T>
struct Vector2
{
    T X;
    T Y;

    Vector2(T x, T y)
        : X{x}, Y{y}
    {
    }
};

// Explicit class template argument: float
Vector2<float> v1{2.0f, 4.0f};

// Compiler deduces the class template argument: float
Vector2 v2{2.0f, 4.0f};

// Also works with 'new'
// 'v3' is a Vector<float>*
auto v3 = new Vector2{2.0f, 4.0f};
```

To help the compiler deduce these arguments, we can write a “deduction guide” to tell it what to do:

```
// Class template
template<class T>
struct Range
{
    // Constructor template
    template<class Pointer>
    Range(Pointer beg, Pointer end)
    {
    }
};

double arr[] = { 123, 456 };

// Compiler error: can't deduce T (class template
argument) from constructor
Range range1{&arr[0], &arr[1]};

// Deduction guide tells the compiler how to deduce the
class template argument
template<class T>
Range(T* b, T* e) -> Range<T>;

// OK: compiler uses deduction guide to deduce that T is
'double'
Range range2{&arr[0], &arr[1]};
```

As we see in this example, deduction guides are written like a function template with the “trailing return syntax.” The major difference is that their name is the name of a class template and their “return type” is a class template with its arguments passed.

Specialization

So far, all of our templates have been instantiated the same way regardless of the template arguments provided to them. Sometimes we want to use an alternate version of the template when certain arguments are provided. This is called *specialization* of a template. Consider this class template:

```
// A very generalized vector
template<typename T, int N>
struct Vector
{
    T Components[N];

    T Dot(const Vector<T, N>& other) const noexcept
    {
        T result{};
        for (int i = 0; i < N; ++i)
        {
            result += Components[i] *
other.Components[i];
        }
        return result;
    }
};

// Usage
Vector<float, 2> v1{2, 4};
DebugLog(v1.Components[0], v1.Components[1]); // 2, 4
```

```
Vector<float, 2> v2{6, 8};
DebugLog(v1.Dot(v2)); // 44
```

Now let's specialize `Vector` for a common use case: two `float` components.

```
// Specialization of the Vector template
template<> // Takes no arguments
struct Vector<float, 2> // Arguments are provided by the specialization instead
{
    // Specialization can have very different contents
    // This union allows access either by the Components array or X and Y
    union
    {
        float Components[2];
        struct
        {
            float X;
            float Y;
        };
    };

    float Dot(const Vector<float, 2>& other) const noexcept
    {
```

```
        // Specialized version doesn't need a loop
        // Easier for readers to understand
        // Compiler can't fail to optimize out the loop
        return X*other.X + Y*other.Y;
    }
};

// We can use X and Y or the Components array to access
the components
Vector<float, 2> v1{2, 4};
DebugLog(v1.Components[0], v1.Components[1]); // 2, 4
DebugLog(v1.X, v1.Y); // 2, 4

// Dot still works
Vector<float, 2> v2{6, 8};
DebugLog(v1.Dot(v2)); // 44
```

There are several reasons we might want to specialize the `Vector` template for common types and sizes of vectors. Perhaps we've inspected the assembly and realized that the compiler didn't optimize out the loop in `Dot`. Perhaps we want to add the convenience of `X` and `Y` data members as synonyms for the first two elements of the `Components` array. Perhaps we want to use SIMD instructions that only work on particular numbers of particular data types. We'll see how to do that later in the book.

Regardless of our reasons, there are a couple aspects of the above example to take note of. First, we're able to specialize not just type parameters like `T` but also non-type parameters like `N`.

Second, our specialization is also named `Vector`. It doesn't get a unique name like `Vector2`. Usually, specializations are meant to be transparent to the user of the template. The template author often provides them to optimize some use case or to provide a superset of functionality in some particular case. The `Vector<float, 2>` specialization *could* have omitted the `Components` array, but then a `Vector<float, 2>` wouldn't be compatible with other instantiations of `Vector`:

```
template<>
struct Vector<float, 2>
{
    // No Components
    float X;
    float Y;

    float Dot(const Vector<float, 2>& other) const
noexcept
    {
        return X*other.X + Y*other.Y;
    }
};

Vector<float, 2> v1{2, 4};

// Compiler error: Vector<float, 2> doesn't have a
Components data member
DebugLog(v1.Components[0], v1.Components[1]); // 2, 4
```

```
// OK: Vector<float, 2> has X and Y
DebugLog(v1.X, v1.Y); // 2, 4
```

That said, sometimes incompatibility is desirable. Take the cross product, for example. We may want to omit this from specializations of 2D vectors as the operation doesn't make a lot of sense. Then again, we might want to return a 3D vector such as `(0, 0, 1)` or `(0, 0, -1)`. Template specializations give us the flexibility to make this design choice.

Finally, we also have the option to "partially specialize" a template. We use a "partial specialization" when we only want to specialize some of the template arguments, not all of them like above. For example, we might want to specialize for 2D vectors but not for `float`:

```
// Partial specialization of the Vector template
// Now takes only one parameter: the type T
template<typename T>
// Pass arguments to the main Vector template
// They can be either parameters to the specialization or
regular arguments
struct Vector<T, 2>
{
    union
    {
        // We can still use T, but we also know that N is
2
        T Components[2];
        struct
```

```
        {
            T X;
            T Y;
        };
    };

    T Dot(const Vector<T, 2>& other) const noexcept
    {
        // The loop is removed, but we still support any
arithmetic type
        return X*other.X + Y*other.Y;
    }
};

// X and Y are available
Vector<float, 2> v1{2, 4};
DebugLog(v1.X, v1.Y); // 2, 4

// Multiple types (float and double) are usable now
Vector<double, 2> v2{6, 8};
DebugLog(v2.X, v2.Y); // 6, 8
```

Conclusion

Both C# and C++ support argument deduction in their generics and templates. As usual, C++ goes way further and with more complexity. It can deduce non-type parameters and template parameters as well as class arguments leading to much more terse code: `Dictionary`, not `Dictionary<MyKeyType, MyValueType>`. Deduction guides give us a tool to really push what's deductible rather than settling for defaults.

28. Variadic Templates

Parameter Packs

A “variadic template” is one that has a “parameter pack.” A parameter pack represents zero or more parameters, just like `params` to a C# function represents an array of zero or more parameters.

Here’s a variadic function template that includes one parameter pack:

```
template<typename ...TArgs>
void LogAll(TArgs... args)
{
}
```

`TArgs` is a parameter pack because it has `...` before the (optional) parameter name: `TArgs`. It’s a parameter pack of type parameters because it starts with `typename`.

To use the parameter pack, we add `...` *after* the name of the parameter: `TArgs...`. The compiler expands this to a comma-delimited list of the arguments.

Let’s look at some instantiations of this template to see how this expansion works:

```
// Zero arguments to the TArgs parameter pack
LogAll();
void LogAll() {}
```

```
// One argument to the TArgs parameter pack
LogAll<int>(123);
void LogAll(int) {}

// One argument to the TArgs parameter pack (with
deduction)
LogAll(123);
void LogAll(int) {}

// Two arguments to the TArgs parameter pack
LogAll(123, 3.14f);
void LogAll(int, float) {}
```

We're free to mix parameter packs with other template parameters:

```
template<typename TPrefix, typename ...TArgs>
void LogAll(TPrefix prefix, TArgs... args)
{
}
```

Unlike C# `params`, the parameter pack doesn't even have to be the last parameter as long as the compiler can deduce all the parameters:

```
// Parameter pack is not the last parameter
template<typename ...TLogParts, typename TPrefix>
void LogWithPrefix(TPrefix prefix, TLogParts... parts)
{
```

```
}

// Compiler deduces that TPrefix is 'float' and TLogParts
is (int, int, int)
LogWithPrefix(3.14f, 123, 456, 789);
```

Note that the compiler can never deduce this with class templates, so the parameter pack must come at the end.

Pack Expansion

Now that we know how to declare packs of template parameters and how to use them in function parameters, let's look at some more ways to use them. One common way is to pass them as function arguments:

```
template<typename ...TArgs> // Template parameter pack
void LogError(TArgs... args) // Use parameter pack to
declare parameters
{
    DebugLog("ERROR", args...); // Pass parameters as
arguments to a function
}

// Pass arguments to function template
// Template arguments deduced from parameter types
LogError(3.14, 123, 456, 789); // ERROR, 3.14, 123, 456,
789

// The compiler instantiates this function
void LogError(double arg1, int arg2, int arg3, int arg4)
{
    DebugLog("ERROR", arg1, arg2, arg3, arg4);
}
```

In this example we passed the arguments straight through as `args...`. This was expanded to `arg1, arg2, arg3, arg4`. If we apply

some operation to the parameter pack name, it'll be applied to all of the arguments:

```
template<typename ...TArgs>
void LogPointers(TArgs... args)
{
    // Apply dereferencing to each value in the pack
    DebugLog(*args...);
}

// Pass pointers
float f = 3.14f;
int i1 = 123;
int i2 = 456;
LogPointers(&f, &i1, &i2); // 3.14, 123, 456

// The compiler instantiates this function
void LogPointers(float* arg1, int* arg2, int* arg3)
{
    DebugLog(*arg1, *arg2, *arg3);
}
```

If we name more than one parameter pack in the same expansion, they get expanded simultaneously:

```
// Class template with two parameters
template<typename T1, typename T2>
struct KeyValue
```

```
{
    T1 Key;
    T2 Value;
};

// Class template with a parameter pack
template<typename ...Types>
struct Map
{
    // ...implementation
};

// Class template with a parameter pack
template<class ...Keys>
struct MapOf
{
    // Member class template with a parameter pack
    template<class ...Values>
    // Derives from Map class template
    // Pass KeyValue<Keys, Values>... as the template arguments to Map
    // Expands to (KeyValue<Keys1, Values1>, KeyValue<Keys2, Values2>, etc.)
    struct KeyValues : Map<KeyValue<Keys, Values>...>
    {
    };
};
```

```
// Instantiate the template with Keys=(int, float) and
Values=(double, bool)
// Pairs derives from Map<KeyValue<int, double>,
KeyValue<float, bool>>
MapOf<int, float>::KeyValues<double, bool> map;

// The compiler instantiates this class
struct MapOf
{
    struct KeyValues : Map<KeyValue<int, double>,
KeyValue<float, bool>>
    {
    };
};
```

Where Packs Can Be Expanded

So far we've seen packs expanded into function parameters, function arguments, and template arguments. There are quite a few more places they can be expanded. First, when initializing with parentheses:

```
struct Pixel
{
    int X;
    int Y;

    Pixel(int x, int y)
        : X(x), Y(y)
    {
    }
};

// Function template takes a parameter pack of ints
template<int ...Components>
Pixel MakePixel()
{
    // Expand into parentheses initialization
    return Pixel(Components...);
};

Pixel pixel = MakePixel<2, 4>();
DebugLog(pixel.X, pixel.Y); // 2, 4
```

Or initializing with curly braces:

```
// Function template takes a parameter pack of ints
template<int ...Components>
Pixel MakePixel()
{
    // Expand into curly braces initialization
    return Pixel{Components...};
};
```

Second, we can expand type parameter packs into packs of non-type parameters:

```
// Class template with a pack of type parameters
template<typename... Types>
struct TypedPrinter
{
    // Function template with a pack of non-type
parameters
    // Formed from the expansion of the Types pack
    template<Types... Values>
    static void Print()
    {
        // Expand the non-type parameters pack
        DebugLog(Values...);
    }
};
```

```
// Instantiate the templates with type and non-type
parameters
TypedPrinter<char, int>::Print<'c', 123>(); // c, 123

// Compiler error: 'c' is not a bool
TypedPrinter<bool, int>::Print<'c', 123>();
```

Third, a class can inherit from zero or more base classes by expanding a pack of types:

```
struct VitalityComponent
{
    int Health;
    int Armor;
};

struct WeaponComponent
{
    float Range;
    int Damage;
};

struct SpeedComponent
{
    float Speed;
};

template<class... TComponents>
```

```
// Expand a pack of base classes
class GameEntity : public TComponents...
{
};

// turret is a class that derives from VitalityComponent
and WeaponComponent
GameEntity<VitalityComponent, WeaponComponent> turret;
turret.Health = 100;
turret.Armor = 200;
turret.Range = 10;
turret.Damage = 15;

// civilian is a class that derives from
VitalityComponent and SpeedComponent
GameEntity<VitalityComponent, SpeedComponent> civilian;
civilian.Health = 100;
civilian.Armor = 200;
civilian.Speed = 2;
```

Fourth, the list of a lambda's captures can be formed by pack expansion:

```
template<class ...Args>
void Print(Args... args)
{
    // Expand the 'args' pack into the lambda capture
list
```

```
    auto lambda = [args...] { DebugLog(args...); };
    lambda();
}

Print(123, 456, 789); // 123, 456, 789
```

Fifth, the `sizeof` operator has a variant that takes a parameter pack. This evaluates to the number of elements in the pack, regardless of their sizes:

```
// General form of summation
// Declaration only since it's never actually
instantiated
template<typename ...TValues>
int Sum(TValues... values);

// Specialization for when there is at least one value
template<typename TFirstValue, typename ...TValues>
int Sum(TFirstValue firstValue, TValues... values)
{
    // Expand pack into a recursive call
    return firstValue + Sum(values...);
}

// Specialization for when there are no values
template<>
int Sum()
{
```

```
    return 0;
}

template<typename ...TValues>
int Average(TValues... values)
{
    // Expand pack into a Sum call
    // Use sizeof... to count the number of parameters in
the pack
    return Sum(values...) / sizeof...(TValues);
}

DebugLog(Average(10, 20)); // 15
```

In this example, the compiler instantiates these templates:

```
// Instantiated for Sum(10, 20)
int Sum2(int firstValue, int value)
{
    // Expand pack into a recursive call
    return firstValue + Sum1(value);
}

// Instantiated for Sum(20)
int Sum1(int firstValue)
{
    return firstValue + Sum0();
}
```

```
// Instantiated for Sum()
int Sum0()
{
    return 0;
}

int Average(int value1, int value2)
{
    return Sum2(value1, value2) / 2;
}

DebugLog(Average(10, 20)); // 15
```

Compiler optimizations will almost always boil this down to a constant:

```
DebugLog(15); // 15
```

Or for arguments `x` and `y` that aren't compile-time constants:

```
DebugLog((x + y) / 2);
```

Conclusion

Variadic templates enable us to write templates based on arbitrary numbers of parameters. This saves us from needing to write nearly-identical versions of the same templates over and over. For example, C# has `Action<T>`, `Action<T1,T2>`, `Action<T1,T2,T3>`, all the way up to `Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>`! The same massive duplication is applied to its `Func` counterpart: `Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>`. This is so painful to write that we usually just don't bother or write a code generator to output all this redundant C#. At no point do we end up with a solution that takes arbitrary numbers of parameters, just *arbitrary enough for now* numbers of parameters.

29. Template Constraints

Constraints

C# has 11 specific `where` constraints we can put on type parameters to generics. These include constraints like `where T : new()` indicating that `T` has a public constructor that takes no parameters. In contrast, C++ provides us with tools to build our own constraints out of compile-time expressions.

So far in C++, all of our templates have had no constraints. Still, we've been able to use those parameters in a great many ways. The difference between the two languages is that the default in C# is to treat generic parameters as the least common denominator type: `System.Object`/`object`. The default in C++ is the other end of the spectrum: template parameters may be used in any way that compiles.

Both languages allow us to set constraints to move the requirements for parameters more toward the opposite end of the spectrum. This means that C# `where` constraints make generics' type parameters *more* specific and therefore allow us to use more specific functionality like calling a constructor with no parameters. We're about to see how C++ template constraints make template parameters *less* specific to perform better overload resolution and give more programmer-friendly compiler error messages. All of this was added in C++20 and is becoming available in all the major compilers.

Requires Clauses

While C# uses the keyword `where` to add a constraint, C++ uses `requires`. Let's jump right in and add one to a function template:

```
struct Vector2
{
    float X;
    float Y;
};

// Variable template
// "Default" value is false
template <typename T>
constexpr bool IsVector2 = false;

// Specialization of the variable template
// Change value to true for a specific type
template <>
constexpr bool IsVector2<Vector2> = true;

// Function template
template <typename TVector, typename TComponent>
// Requires clause
// Compile-time expression evaluates to a bool
// Can use template parameters here
requires IsVector2<TVector>
// The function
```

```
TComponent Dot(TVector a, TVector b)
{
    return a.X*b.X + a.Y*b.Y;
}

// OK
Vector2 vecA{2, 4};
Vector2 vecB{2, 4};
DebugLog(Dot<Vector2, float>(vecA, vecB));

// Compiler error:
//
// Candidate template ignored: constraints not satisfied
// [with TVector = int, TComponent = int]
// TComponent Dot(TVector a, TVector b)
//            ^
// test.cpp:60:10: note: because 'IsVector2<int>'
evaluated to false
// requires IsVector2<TVector>
DebugLog(Dot<int, int>(2, 4));
```

Rather than a language-specified `where` constraint like we'd use in C#, we instead specify any compile-time expression after the keyword `requires`. In this case we're using a variable template that we've defaulted to `false` and then specialized to opt-in the specific `Vector2` type to `true`.

These compile-time expressions have access to the template parameters. Like other compile-time expressions, they're allowed to

be arbitrarily complex. For example, consider this variable template that doesn't have a `requires` clause:

```
// Variable template
template <typename T, int N>
// Recurse to next-lower value
constexpr T SumUpToN = N + SumUpToN<T, N-1>;

// Specialization for 0 stop recursion
template <typename T>
constexpr T SumUpToN<T, 0> = 0;

// OK
DebugLog(SumUpToN<float, 3>); // 6

// Compile error:
//
// test.cpp:44:28: fatal error: recursive template
instantiation exceeded
// maximum depth of 1024
// constexpr T SumUpToN = N + SumUpToN<T, N-1>;
//                            ^
// test.cpp:44:28: note: in instantiation of variable
template specialization
// 'SumUpToN<float, -1025>' requested here
// test.cpp:44:28: note: in instantiation of variable
template specialization
// 'SumUpToN<float, -1024>' requested here
```

```
// test.cpp:44:28: note: in instantiation of variable
template specialization
// 'SumUpToN<float, -1023>' requested here
// test.cpp:44:28: note: in instantiation of variable
template specialization
// 'SumUpToN<float, -1022>' requested here
// test.cpp:44:28: note: in instantiation of variable
template specialization
// 'SumUpToN<float, -1021>' requested here
//
// ... many more lines of errors
DebugLog(SumUpToN<float, -1>);
```

To stop that infinite recursion, we can add a `requires` clause:

```
template <typename T, int N>
// Constraint to positive values only
requires (N >= 0)
constexpr T SumUpToN = N + SumUpToN<T, N-1>;

template <typename T>
constexpr T SumUpToN<T, 0> = 0;

// OK
DebugLog(SumUpToN<float, 3>); // 6

// Compiler error:
//
```

```
// test.cpp:54:14: error: constraints not satisfied for
variable template
// 'SumUpToN' [with T = float, N = -1]
//     DebugLog(SumUpToN<float, -1>);
//              ^~~~~~~~~~~~~~~~~~
// test.cpp:42:11: note: because '-1 >= 0' (-1 >= 0)
evaluated to false
// requires (N >= 0)
DebugLog(SumUpToN<float, -1>);
```

We've successfully stopped the infinite recursion before it started with a `requires` constraint. The compiler didn't need to instantiate thousands of templates and it didn't print out thousands of error messages for us to decipher. Instead, we simply get one readable message telling us that the constraint wasn't satisfied.

Concepts

So far `requires` has been a pretty limited tool. That's because this isn't the primary way of using it. Instead, we usually use `requires` to define a "concept." A concept is supposed to be a semantic description of a category of types. For example, we might define a `Number` concept as a type that supports various numeric operators:

```
template <typename T>
concept Number = requires(T t) {
    t + t;
    t - t;
    t * t;
    t / t;
    -t;
    +t;
    --t;
    ++t;
    t--;
    t++;
};
```

This is a new use of `requires` in a couple of ways. First, we've added a parameter list like a function has. This gives us a named parameter `t` with the type `T` matching the template parameter. Second, we then use that parameter in a book of statements, similar to what a function is. If those statements compile, the constraint is satisfied.

Finally, we save the constraint created by `requires` using the `concept` keyword. This gives us a way to name the constraint elsewhere when we want to use it. All concepts are templates taking type parameters since their purpose is to categorize types.

Now let's put the concept to use:

```
// Function template with two type parameters (the latter
is defaulted)
template <typename TVal, typename TThreshold=TVal>
// Requires clause names concepts
requires Number<TVal> && Number<TThreshold>
// The function
bool IsNearlyZero(TVal val, TThreshold threshold)
{
    return (val < 0 ? -val : val) < threshold;
}

// All of these are OK since double, float, and int
satisfy Number
DebugLog(IsNearlyZero(0.0, 0.1)); // true
DebugLog(IsNearlyZero(0.2f, 0.1f)); // false
DebugLog(IsNearlyZero(2, 1)); // true

struct Player{};

// Compiler error: Player doesn't satisfy the Number
constraint
DebugLog(IsNearlyZero(Player{}, Player{}));
```

Instead of directly evaluating to a `bool`, this use of `requires` names the `Number` concept and passes arguments to it: `Number<TVal>` and `Number<TThreshold>`. The `requires` clause can still perform binary logic using `&&` and `||` operators to combine concepts or check `bool` values.

There are a few alternate syntaxes we can use. First, we can put the `requires` clause after the parameter list:

```
template <typename TVal, typename TThreshold=TVal>
bool IsNearlyZero(TVal val, TThreshold threshold)
    requires Number<TVal> && Number<TThreshold>
{
    return (val < 0 ? -val : val) < threshold;
}
```

If we have a trivial `requires` clause that simply names a single concept, which is quite typical, we can replace `typename` with the name of the concept:

```
// Trivial concept version (still using typename)
template <typename T>
bool IsNearlyZero(T val, T threshold)
    requires Number<T>
{
    return (val < 0 ? -val : val) < threshold;
}

// Replace typename with concept name
```

```
template <Number T>
bool IsNearlyZero(T val, T threshold)
{
    return (val < 0 ? -val : val) < threshold;
}
```

If we're using `auto` parameters to create "abbreviated function templates," then we won't have the `template`. Instead, we can simply put the concept name before `auto` to require that the parameter satisfy that template:

```
bool IsNearlyZero(Number auto val, Number auto threshold)
{
    return (val < 0 ? -val : val) < threshold;
}
```

Regardless of which syntax we choose, the usage is always the same as the above because all of these are equivalent. The preference of syntax is mostly a matter of the level of flexibility we need and of style.

Because this form of `requires` defines a concept and a concept is what's named after the other form of `requires`, we sometimes use `requires requires` to define an ad-hoc concept:

```
template <typename T>
// Requires clause names ad-hoc concept
requires requires(T t) {
    t + t;
```

```
    t - t;
    t * t;
    t / t;
    -t;
    +t;
    --t;
    ++t;
    t--;
    t++;
}
bool IsNearlyZero(T val, T threshold)
{
    return (val < 0 ? -val : val) < threshold;
}
```

Note that this `Number` concept is quite incomplete and for example purposes only. The C++ Standard Library has many well-designed concepts such as `std::integral` and `std::floating_point` that are suitable for production code.

Combining Concepts

Many concepts are defined in terms of other concepts. Just like how we used `&&` in our `requires` clause, we can do the same when defining a `concept`:

```
// Define a concept in terms of another concept and an
ad-hoc concept
template <typename T>
concept Integer = Number<T> && requires(T t) {
    t << t;
    t <<= t;
    t >> t;
    t >>= t;
    t % t;
};
```

We can also use concepts within the definition of our concepts. These are known as “nested concepts:”

```
template <typename T>
concept Vector2 = requires(T t) {
    // Use a concept from within a concept
    // This requires the type of t.X to satisfy the
Number constraint
    Number<decltype(t.X)>;

    // Also require Y to be a Number
```

```
    Number<decltype(t.Y)>;
};

struct Vector2f
{
    float X;
    float Y;
};

bool IsOrthogonal(Vector2 auto a, Vector2 auto b)
{
    return (a.X*b.X + a.Y*b.Y) == 0;
}

Vector2f a{0, 1};
Vector2f b{1, 0};
Vector2f c{1, 1};
DebugLog(IsOrthogonal(a, b)); // true
DebugLog(IsOrthogonal(a, c)); // false
```

Alternately, we can use “compound requirements” to implicitly pass the type that an expression evaluates to as the first argument to a concept. Here’s how the `Vector2` concept could have used this:

```
template <typename T>
concept Vector2 = requires(T t) {
    // t.X must evaluate to a type that satisfies the
Number constraint
```

```
    {t.X} -> Number;

    {t.Y} -> Number;
};
```

Overload Resolution

We've seen how to use specialization to write custom versions of templates. This generally works well for particular types such as `float`, but it's difficult to specialize for whole *categories* of types. This is where concepts come in. When calling an overloaded function, the compiler will look at the concepts to find the one that's most constrained:

```
// Incomplete definition of a dynamic array class
struct List
{
    int Length;
    int* Array;

    int* GetBegin()
    {
        return Array;
    }

    int& operator[](int i)
    {
        return Array[i];
    }
};

// Incomplete definition of a linked list class
struct LinkedList
{
```

```
    struct Node
    {
        int Value;
        Node* Next;
    };

    Node* Head;

    Node* GetBegin()
    {
        return Head;
    }
};

// A concept that defines types that can be iterated
template <typename T>
concept Iterable = requires(T t) {
    t.GetBegin();
};

// A concept that defines types that can be indexed into
// This is more constrained than just Iterable
template <typename T>
concept Indexable = Iterable<T> && requires(T t) {
    t[0]; // Can read from an index
    t[0] = 0; // Can write to an index
};
```

```
// Indexable overload simply indexes: O(1)
int GetAtIndex(Indexable auto collection, int index)
{
    return collection[index];
}

// Iterable version has to walk the list: O(N)
int GetAtIndex(Iterable auto collection, int index)
{
    auto cur = collection.GetBegin();
    for (int i = 0; i < index; ++i)
    {
        cur = cur->Next;
    }
    return cur->Value;
}

// Overload resolution calls the Indexable version
List list;
int a = GetAtIndex(list, 1000);

// Overload resolution calls the Iterable version
LinkedList linkedList;
int b = GetAtIndex(linkedList, 1000);
```

C# Equivalency

Now that we know how constraints work in C++, let's see how we'd approximate each of the 11 `where` constraints that C# offers. To do this, we'll use some pre-defined concepts out of the C++ Standard Library's `<concepts>` header and a variable template out of `<type_traits>` rather than writing our own versions of these.

C# Constraint	C++ Concept (approximation)
`where T : struct`	`template <class T> concept C = std::is_class_v<T>;`
`where T : class`	`template <class T> concept C1 = !Nullable<T> && std::is_class_v<T>`
`where T : class?`	`template <class T> concept C = std::is_class_v<T>;`
`where T : notnull`	`template <class T> concept C = !Nullable<T>;`
`where T : unmanaged`	N/A. All C++ types are unmanaged.
`where T : new()`	`std::default_initializable<T>`
`where T : BaseClass`	`template <class T> concept C = !Nullable<T> && std::derived_from<T, BaseClass>;`
`where T : BaseClass?`	`std::derived_from<T, BaseClass>`
`where T : Interface`	`template <class T> concept C = !Nullable<T> && std::derived_from<T, BaseClass>;`

C# Constraint	C++ Concept (approximation)
`where T : Interface?`	`std::derived_from<T, BaseClass>`
`where T : U`	`std::derived_from<T, U>`

In the table above, `std::derived_from` and `std::default_initializable` are concepts and `std::is_class_v` is a `bool` variable template. `Nullable` isn't in the C++ Standard Library, but it might look like this:

```
template <class T> concept Nullable =
    // Can assign nullptr to it
    requires(T t) { t = nullptr; } &&
    // Has a user-defined conversion operator to nullptr
or is a pointer
    (requires(T t) { t.operator decltype(nullptr)(); } ||
std::is_pointer_v<T>);
```

This, and some of the other concepts, are approximations of their C# equivalents. C++ doesn't have exact matches for C# "nullable contexts" and other subtle language differences. Feel free to adjust these concepts to suit your intended usage.

Conclusion

C++ constraints provided by `requires` and `concept` fill a similar role to C# constraints provided by `where`. As is often the case when comparing the two languages, the C++ version is essentially a superset of the C# functionality. While some concepts are provided by the C++ Standard Library via the `<concepts>` header, we're also given the tools to write our own concepts as we did above with `Nullable` and others.

The constraints we create allow us to limit what our templates are allowed to work on, express that intent to users of our templates, generate much more readable compiler errors, and even choose the most optimal overloaded function.

This is in contrast to C# constraints that *enable* our generics to use more functionality of their type parameters. Because only 11 basic constraints are provided with no ability for us to create our own constraints, we're often forced into trade-offs such as taking a performance hit due to calling functions on an interface, creating garbage due to boxing to a reference type, or jumping through hoops to write generic code.

30. Type Aliases

Typedef

There are two main ways of creating type aliases in C++. The first, `typedef`, is inherited from C. It's still common to see in C++ codebases, but later in the chapter we'll learn another approach that's essentially a complete replacement for `typedef`.

To create an alias this way, we write `typedef SourceType AliasName;`:

```
// Create an alias of "unsigned int" called "uint32"
typedef unsigned int uint32;

// Use the "uint32" alias in place of "unsigned int"
constexpr uint32 ZERO = 0;
```

C# `using X = Y;` aliases can only appear in two places. If they're placed at the start of a `.cs` file, they're in scope in that file. If they're placed in a `namespace` block, they're in scope in that block. This means they're never usable in other namespace blocks or other files.

C++ type aliases work differently. They can be added to other kinds of scopes and used across files:

```
////////////
// Math.h //
////////////
```

```
namespace Integers
{
    // Add a "uint32" alias for "unsigned int" in the
Integers namespace
    typedef unsigned int uint32;
}

// Use "uint32" like any other member of the Integers
namespace
constexpr Integers::uint32 ZERO = 0;

////////////
// Game.h //
////////////

// Include header file to get access to the Integers
namespace and ZERO
#include "Math.h"

constexpr Integers::uint32 MAX_HEALTH = 100;

//////////////
// Game.cpp //
//////////////

// Include header file to get access to Integers, ZERO,
and MAX_HEALTH
#include "Game.h"
```

```
DebugLog(ZERO); // 0
DebugLog(MAX_HEALTH); // 100

// The type alias is usable here, too
for (Integers::uint32 i = 0; i < 3; ++i)
{
    DebugLog(i); // 0, 1, 2
}
```

This example added a type alias to a namespace, but we can add them to almost any kind of scope. For example, we might add an alias to just a single function:

```
void Foo()
{
    // Type alias scoped to just one function
    typedef unsigned int uint32;

    for (uint32 i = 0; i < 3; ++i)
    {
        DebugLog(i); // 0, 1, 2
    }
}
```

Or even a block within a function:

```
void Foo()
{
    {
        // Type alias scoped to just one function
        typedef unsigned int uint32;

        for (uint32 i = 0; i < 3; ++i)
        {
            DebugLog(i); // 0, 1, 2
        }
    }

    // Compiler error: type alias is only visible in the
above block
    uint32 x = 0;
}
```

It's also common to see type aliases added as members of classes:

```
struct Player
{
    // Player::HealthType is now an alias for "unsigned
int"
    typedef unsigned int HealthType;

    // We can use it here without the namespace qualifier
    HealthType Health = 0;
```

```
};

// We can use it outside of the class by adding the
namespace qualifier
void ApplyDamage(Player& player, Player::HealthType
amount)
{
    player.Health -= amount;
}
```

This approach is particularly useful when we think we might change the type of `Health` later on. We can simply update the `typedef` line to `typedef unsigned long long int HealthType;` and the types of `Health` and `amount` will both be changed. In a larger project, this might save us from having to update hundreds or thousands of types.

It's important to remember that, like C# type aliases, these `typedef` statements don't create new types. When we use `uint32`, it's exactly the same as if we used `unsigned int`. The alias we create is exactly that: another way to refer to the same type.

In addition to the simple `typedef` statements we've used so far, we can also write a couple kinds of more complex statements. First, we can make more than one alias in a single statement. This works similarly to declaring multiple variables at once:

```
// Create four type aliases:
// 1) "Int" for "int"
// 2) "IntPointer" for "int*" a.k.a. "a pointer to an
int"
```

```
// 3) "FunctionPointer" for "int (&)(int, int)"
//    a.k.a. "reference to function that takes two ints
and returns an int"
// 4) "IntArray" for "int[2]" a.k.a "an array of two
ints"
typedef int Int, *IntPointer, (&FunctionPointer)(int,
int), IntArray[2];

Int one = 1;
DebugLog(one); // 1

IntPointer p = &one;
DebugLog(*p); // 1

int Add(int a, int b)
{
    return a + b;
}

FunctionPointer add = Add;
DebugLog(add(2, 3)); // 5

IntArray array = { 123, 456 };
DebugLog(array[0], array[1]); // 123, 456
```

Second, `typedef` is sometimes used to create `struct` types. This is a hold-over from C that's not necessary in C++, but some legacy code may still do this and it's supported for backwards-compatibility reasons. This is valid C and C++:

```
// C code

// Create two type aliases:
// 1) "Player" for "struct { int Health; int Speed; }"
// 2) "PlayerPointer" for "Player*" a.k.a. "pointer to
Player"
typedef struct
{
    int Health;
    int Speed;
} Player, *PlayerPointer;

Player p;
p.Health = 100;
p.Speed = 10;
DebugLog(p.Health, p.Speed); // 100, 10

PlayerPointer pPlayer = &p;
DebugLog(pPlayer->Health, pPlayer->Speed); // 100, 10
```

Without the `typedef`, C code would be forced to prefix `Player` with `struct` like this:

```
// C code

struct Player
{
```

```
    int Health;
    int Speed;
};

struct Player p; // C requires "struct" prefix
p.Health = 100;
p.Speed = 10;
DebugLog(p.Health, p.Speed); // 100, 10
```

Again, neither the `struct` prefix nor the `typedef` workaround are necessary in C++. It's just important to know why `typedef` is used like this since it's still commonly seen in C++ codebases.

Using Aliases

Since C++11, `typedef` is no longer the preferred way of creating type aliases. The new way looks a lot more like C#'s `using X = Y;`. Note that the order of the alias and the type has reversed compared to `typedef`:

```
// Create an alias of "unsigned int" called "uint32"
using uint32 = unsigned int;

// Use the "uint32" alias in place of "unsigned int"
constexpr uint32 ZERO = 0;
```

We're simply listing the type name on the right side. This is particularly more readable than `typedef` for some of the more complex types we've seen since the alias name isn't mixed in with the type being aliased:

```
// Alias for a pointer to an int
using IntPointer = int*;

// Alias for a function that takes two ints and returns
an int
using FunctionPointer = int (*)(int, int);

// Alias for an array of two int elements
using IntArray = int[2];
```

This syntax is exactly equivalent to a `typedef`. Both create an alias to the original type, not a new type. Both can appear in global, namespace, function, or function block scope. Most programmers find this form more readable since it mimics the form of variable assignment and separates the alias from the original type with an `=`.

Multiple aliases can't be created in one statement with `using`. This is probably for the best as that `typedef` syntax is relatively difficult to read and seldom used:

```
// Compiler error: can only create one alias at a time
using uint32 = unsigned int, f32 = float;
```

Besides these syntactic advantages, `using` has an functional improvement as well: we can create alias templates. Consider this code that doesn't make use of alias templates:

```
// Namespace with a class template
namespace Math
{
    template <typename TComponent>
    struct Vector2
    {
        TComponent X;
        TComponent Y;
    };
}

// Another namespace with a class template
namespace Collections
```

```
{
    template <typename TKey, typename TValue>
    struct HashMap
    {
        // ... implementation
    };
}

// Type names start getting long
Collections::HashMap<int32_t, Math::Vector2<float>>
playerLocations;
Collections::HashMap<int32_t, Math::Vector2<int32_t>>
playerScores;

// Shortening requires an alias for each template
instantiation
using vec2f = Math::Vector2<float>;
using vec2i32 = Math::Vector2<int32_t>;
Collections::HashMap<int32_t, vec2f> playerLocations;
Collections::HashMap<int32_t, vec2i32> playerScores;
```

Now consider if we do have access to alias templates:

```
// Template of a type alias
// Takes two type parameters: TKey and TValue
template <typename TKey, typename TValue>
using map = Collections::HashMap<TKey, TValue>; // Can
use parameters in alias
```

```
template <typename TComponent>
using vec2 = Math::Vector2<TComponent>;

// Pass arguments to the aliases just like any other
template
map<int32_t, vec2<float>> playerLocations;
map<int32_t, vec2<int32_t>> playerLocations;

// We can still create non-template type aliases to get
more specific
using vec2f = vec2<float>;
using vec2i32 = vec2<int32_t>;
map<int32_t, vec2f> playerLocations;
map<int32_t, vec2i32> playerScores;

// And even more specific...
using LocationMap = map<int32_t, vec2f>;
using ScoreMap = map<int32_t, vec2i32>;
LocationMap playerLocations;
ScoreMap playerScores;
```

Alias templates give us a tool to keep some of the type parameters without being forced to alias a concrete type. These templates can be reused, as we did with both `map` and `vec`, rather than duplicating aliases. This becomes more and more useful as types become more complicated and generic.

These alias templates inherit all the functionality of other kinds of templates, such as for functions and classes. For example, we can

use non-type parameters:

```
// Class template for a fixed-length array
template <typename TElement, int N>
struct FixedList
{
    int Length = N;
    TElement Elements[N];

    TElement& operator[](int index)
    {
        return Elements[index];
    }
};

// Alias template taking a non-type parameter: int N
template <int N>
using ByteArray = FixedList<unsigned char, N>;

// Pass a non-type argument to the alias template: <3>
ByteArray<3> bytes;
bytes[0] = 10;
bytes[1] = 20;
bytes[2] = 30;
DebugLog(bytes.Length, bytes[0], bytes[1], bytes[2]); //
3, 10, 20, 30
```

Permissions

Lastly, and quickly, there's one final use for type aliases. As we've seen before, class permissions like `private` can be used to prevent code outside of the class from using certain members. This applies to types that the class creates:

```
class Outer
{
    // Member type that is private, the default for
"class"
    struct Inner
    {
        int Val = 123;
    };
};

// Compiler error: Inner is private
Outer::Inner inner;
```

Type aliases can be used to avoid this restriction. This works because the compiler only checks the permissions of the type being used. If that type is an alias for another type, the aliased type's permission is irrelevant and ignored:

```
class Outer
{
    // Member type is still private
```

```
    struct Inner
    {
        int Val = 123;
    };

public:

    // Type alias is public
    using InnerAlias = Inner;
};

// OK: uses permission level of InnerAlias, not Inner
Outer::InnerAlias inner;
```

Usually we'll just specify the desired permission level to begin with. In cases such as using third-party libraries, we don't have the ability to change that original permission level. This workaround can be used to get the access we need.

Conclusion

Type aliases in C++ go way beyond their C# counterparts. They're not limited to a single source code file or namespace block. Instead, we can and commonly do declare them in header files as globals, in namespaces, and as class members. We declare terse names in functions or even blocks in functions to avoid a lot of type verbosity, especially when using generic code such as a `HashMap<TKey, TValue>`. These aliases can be created once and shared across the whole project, not just within one file.

Alias templates go even further by allowing us to create aliases that don't resolve to a concrete type. These can prevent a lot of code duplication and give names to in-between steps such as `map` that lives between the very generic `HashMap<TKey, TValue>` and the very concrete `LocationMap`. They inherit the powers of other C++ templates with capabilities including non-type parameters and variable numbers of parameters.

31. Deconstructing and Attributes

Structured Bindings

“Deconstructing” in C# is the process by which we extract the fields out of a struct or class and into individual variables. It looks like this:

```
// C#

// A type we want to deconstruct
struct Vector2
{
    public float X;
    public float Y;

    // Create a Deconstruct method that takes an 'out' param for each variable
    // to deconstruct into and returns void
    public void Deconstruct(out float x, out float y)
    {
        x = X;
        y = Y;
    }
}

// Instantiate the deconstructable type
var vec = new Vector2{X=2, Y=4};
```

```
// Deconstruct. Implicitly calls vec.Deconstruct(out x,
out y).
// x is a copy of vec.X and y is a copy of vec.Y
var (x, y) = vec;

DebugLog(x, y); // 2, 4
```

In C++ terminology, we don't "deconstruct" but rather "create structured bindings." Here's the equivalent code:

```
// A type we want to create structured bindings for
struct Vector2
{
    float X;
    float Y;
};

// Instantiate that type
Vector2 vec{2, 4};

// Create structured bindings. x is a copy of vec.X and y
is a copy of vec.Y.
auto [x, y] = vec;

DebugLog(x, y); // 2, 4
```

So far it's essentially the same in the two language except for two changes. First, C++ uses square brackets (`[x, y]`) instead of

parentheses (`(x, y)`). Second, C++ doesn't require us to write a `Deconstruct` function. Instead, the compiler simply uses the declaration order of the fields of `Vector2` so that `x` lines up with `X` and `y` with `Y`. This mirrors initialization where `Vector2{2, 4}` initializes the data members in declaration order: `X` then `Y`.

This can be customized to by "tuple-like" types, but it takes more work than in C#:

```
struct Vector2
{
    float X;
    float Y;

    // Get a data member of a const vector
    template<std::size_t Index>
    const float& get() const
    {
        // Assert the only two valid indices
        static_assert(Index == 0 || Index == 1);

        // Return the right one based on the index
        if constexpr(Index == 0)
        {
            return X;
        }
        return Y;
    }

    // Get a data member of a non-const vector
```

```
    template <std::size_t Index>
    float& get()
    {
        // Cast to const so we can call the const
overload of this function
        // to avoid code duplication
        const Vector2& constThis = const_cast<const
Vector2&>(*this);

        // Call the const overload of this function
        // Returns a const reference to the data member
        const float& constComponent =
constThis.get<Index>();

        // Cast the data member to non-const
        // This is safe since we know the vector is non-
const
        float& nonConstComponent = const_cast<float&>
(constComponent);

        // Return the non-const data member reference
        return nonConstComponent;
    }
};

// Specialize the tuple_size class template to derive
from integral_constant
// Pass 2 since Vector2 always has 2 components
```

```
template<>
struct std::tuple_size<Vector2> :
std::integral_constant<std::size_t, 2>
{
};

// Specialize the tuple_element struct to indicate that
index 0 of Vector2 has
// the type 'float'
template<>
struct std::tuple_element<0, Vector2>
{
    // Create a member named 'type' that is an alias for
'float'
    using type = float;
};

// Same for index 1
template<>
struct std::tuple_element<1, Vector2>
{
    using type = float;
};

// Usage is the same
Vector2 vec{2, 4};
auto [x, y] = vec;
DebugLog(x, y);
```

The result of the above code is that `Vector2` is now a “tuple-like” type, usable with structured bindings and several generic algorithms of the C++ Standard Library.

The final kind of object we can create structured bindings for is an array. This isn’t allowed by default in C#:

```
// Create an array of two int elements
int arr[] = { 2, 4 };

// Create structured bindings. x is a copy of arr[0] and
y is a copy of arr[1].
auto [x, y] = arr;

DebugLog(x, y); // 2, 4
```

It’s important to note that the structured bindings we’ve been creating *have* to be automatically-typed with `auto`. Even if we know the type, we can’t use it:

```
int arr[] = { 2, 4 };

// Compiler error: must use the 'auto' type here
int [x, y] = arr;
```

What we are allowed to do is create structured binding references with `auto&`:

```
int arr[] = { 2, 4 };

// Create copies of arr[0] and arr[1]
auto [xc, yc] = arr;

// Create references to arr[0] and arr[1]
auto& [xr, yr] = arr;

// Modify the elements of the array
arr[0] = 20;
arr[1] = 40;

DebugLog(xc, yc); // 2, 4
DebugLog(xr, yr); // 20, 40
```

The same is also true for rvalue references via `auto&&`:

```
Vector2 Add(Vector2 a, Vector2 b)
{
    return Vector2{a.X+b.X, a.Y+b.Y};
}

// Compiler error: return value of Add isn't an lvalue so
can't take lvalue ref
auto& [x, y] = Add(Vector2{2, 4}, Vector2{6, 8});

// OK
```

```
auto&& [x, y] = Add(Vector2{2, 4}, Vector2{6, 8});
DebugLog(x, y); // 8, 12
```

Both forms of references as well as copies may also be `const`:

```
Vector2 vec{2, 4};

// Constant copy
const auto [xc, yc] = vec;

// Constant lvalue reference
const auto& [xr, yr] = vec;

// Constant rvalue reference
const auto&& [xrr, yrr] = Add(Vector2{2, 4}, Vector2{6,
8});
```

We can also use other forms of initialization when creating structured bindings. So far we've copy-initialized our variables with `=` but we can also direct-initialize them with `{}` and `()`:

```
Vector2 vec{2, 4};

// Copy-initialize
auto [x1, y1] = vec;

// Direct-initialize with curly braces
auto [x2, y2]{vec};
```

```
// Direct-initialize with parentheses
auto [x3, y3](vec);
```

Finally, C# supports ignoring some deconstructed variables with “discards” in the form of `_`. This isn’t supported in C++, so we’ll need to explicitly name and ignore them to avoid a compiler warning:

```
Vector2 vec{2, 4};

auto [x, y] = vec;
static_cast<void>(x); // One way of explicitly ignoring x
(void)x; // Another way of ignoring x

// Use only y
DebugLog(y); // 4
```

Attributes

C# attributes associate some metadata with an entity like a class, a method, or a parameter. This metadata can be used in one of two ways. First, the compiler can query attributes like `[MethodImplAttribute(MethodImplOptions.AggressiveInlining)]` to modify how compilation works. Second, our C# code can query attributes at runtime via reflection to make use of it in various custom ways.

As C++ doesn't support reflection, it's usage of attributes doesn't cover the runtime use cases. It does, however, cover the compile-time use cases. Because this functionality is implemented by the compiler, not us, we can't create custom attributes. Instead, we use two built-in sets of attributes. First, there are several attributes defined by the C++ language:

Attribute	**Version**	**Meaning**
`[[noreturn]]`	C++11	This function will never return
`[[carries_dependency]]`	C++11	Unnecessary memory fence instructions for this can be removed in some situations
`[[deprecated]]`	C++14	This is deprecated
`[[deprecated("why")]]`	C++14	This is deprecated for a specific reason
`[[fallthrough]]`	C++17	This `case` in a `switch` intentionally falls through to the next `case`
`[[nodiscard]]`	C++17	A compiler warning should be generated if this is ignored

Attribute	Version	Meaning
`[[nodiscard("msg")]]`	C++20	A compiler warning with a particular message should be generated if this is ignored
`[[maybe_unused]]`	C++17	No compiler warning should be generated for not using this
`[[likely]]`	C++20	This branch is likely to be taken
`[[unlikely]]`	C++20	This branch is unlikely to be taken
`[[no_unique_address]]`	C++20	This non-static data member doesn't need to have a unique memory address

There are two aspects of these to take notice of. First, and trivially, C++ attributes use two square brackets (`[[X]]`) instead of one in C# (`[X]`). Second, all of the attributes are for one of two purposes: controlling compiler warnings and optimizing generated code.

The second set of attributes is compiler-specific and not part of the C++ standard. Clang, for example, has a ton of them for various purposes. If these are ever specified and the compiler doesn't support them, the're simply ignored.

Now let's look at some code that uses them:

```
class File
{
    FILE* handle = nullptr;
```

```
public:

    ~File()
    {
        if (handle)
        {
            ::fclose(handle);
        }
    }

    // Generate a compiler warning if the return value is
ignored
    [[nodiscard]] bool Close()
    {
        if (!handle)
        {
            return true;
        }
        return ::fclose(handle) == 0;
    }

    // Generate a compiler warning if the return value is
ignored
    [[nodiscard]] bool Open(const char* path, const char*
mode)
    {
        if (!handle)
```

```
        {
            // No compiler warning because return value
is used
            if (!Close())
            {
                return false;
            }
        }
        handle = ::fopen(path, mode);
        return handle != nullptr;
    }
};

File file{};

// Compiler warning: return value ignored
file.Open("/path/to/file", "r");

// Compiler warning: unused variable
bool success = file.Open("/path/to/file", "r");

// No compiler warning: suppress the unused variable
[[maybe_unused]] bool success =
file.Open("/path/to/file", "r");

// No compiler warning because return value is used
if (!file.Open("/path/to/file", "r"))
{
```

```
    DebugLog("Failed to open file");
}
```

To use more than one attribute at a time, add commas like when declaring multiple variables at a time:

```
// Both [[deprecated("why")]] and [[nodiscard]]
[[deprecated("Wasn't very good. Use Hash2() instead."),
nodiscard]]
uint32_t Hash(const char* bytes, std::size_t size)
{
    uint32_t hash = 0;
    for (std::size_t i = 0; i < size; ++i)
    {
        hash += bytes[i];
    }
    return hash;
}
```

For attributes in namespaces, such as provided by compilers, we use the familiar scope resolution operator (`::`) to refer to them:

```
// Use the "nodebug" attribute in the "gnu" namespace
// Do not generate debugging information for this
function
[[gnu::nodebug]]
float Madd(float a, float b, float c)
{
```

```
    return a * b + c;
}
```

When using multiple attributes in namespaces, there's a shortcut since C++17 that avoids duplicating the namespace name:

```
// Both [[gnu::returns_nonnull]] and [[gnu::nodebug]]
[[using gnu: returns_nonnull, nodebug]]
void* Allocate(std::size_t size)
{
    if (size < 4096)
    {
        return SmallAlloc(size);
    }
    return BigAlloc(size);
}
```

Attributes can appear in a great many places in C++: variables, functions, names, blocks, return values, and so forth. If adding the attribute makes logical sense, it's probably allowed.

Conclusion

C++ structured bindings are a different take on C#'s deconstructing. We don't need to write any code at all to create structured bindings for structs and arrays. For tuple-like types, we have to write quite a bit more than a `Deconstruct` method in C#. Thankfully, that's rarely needed due to the presence of the `std::tuple` class template in the Standard Library which is obviously tuple-like and therefore supports deconstructing.

C++ attributes are one of the rare areas of the language that's actually *less* powerful than its C# counterpart. It fulfills compile-time purposes such as by controlling warnings and optimization, but it doesn't support any run-time use cases due to the lack of reflection in the language. Third-party libraries (example) are required to add on run-time reflection if needed, but they're not integrated into the core language. This may change in C++23 or another future version as there has been much work on integrating compile-time reflection into the language.

32. Thread-Local Storage and Volatile

Thread-Local Storage

Thread-Local Storage is a way of storing one variable per thread. Both C# and C++ have support for this. In C#, we add the `[ThreadStatic]` attribute to a `static` field. A common bug results from the field's initializer being run only once, like other `static` fields, not once per thread.

```
// C#
public class Counter
{
    // One int stored per thread
    // Initialized once, not one per thread
    [ThreadStatic] public static int Value = 1;
}

Action a = () => DebugLog(Counter.Value);
Thread t1 = new Thread(new ThreadStart(a));
Thread t2 = new Thread(new ThreadStart(a));
t1.Start();
t2.Start();
t1.Join();
t2.Join();

// First thread runs and the first use of Counter
initializes Value to 1
// Second thread runs and doesn't initialize Value. Uses
```

```
the default of 0.
// Output: 1 then 0
```

C++ uses the keyword `thread_local` instead of an attribute. This keyword can be applied to static data members like in C#. It can also be applied to variables at global scope, namespace scope, or any level of block scope:

```
// Global variable
thread_local int global = 1;

namespace Counters
{
    // Namespace variable
    thread_local int ns = 1;
}

struct Counter
{
    // Static data member
    // Inline initialization isn't allowed for non-const
static data members
    static thread_local int member;
};
// Initialization outside the class is OK
thread_local int Counter::member = 1;

void Foo()
```

```
{
    // Local variable
    thread_local int local = 1;

    {
        // Variable in any nested block
        thread_local int block = 1;
    }
}
```

Additionally, `thread_local` variables can be marked `static` or `extern` to control linkage:

```
// Globals can be static or extern
static thread_local int global1 = 1;
extern thread_local int global2 = 2;

namespace Counters
{
    // Namespace variables can be static or extern
    static thread_local int ns1 = 1;
    extern thread_local int ns2 = 2;
}

void Foo()
{
    // Local variables can be static, but not extern
    static thread_local int local = 1;
```

```
    {
        // Nested block variables can be static, but not
extern
        static thread_local int block = 1;
    }
}
```

Note that `static` doesn't affect their storage duration. All `thread_local` variables are allocated and initialized when the thread begins. The exact order of initialization isn't specified, so it shouldn't be relied on. This is a change from C# where the intialization doesn't occur per-thread at all.

```
// Initialized for each thread, not just once as in C#
static thread_local int counter = 1;

auto a = []{ DebugLog(counter); };
std::thread t1{a};
std::thread t2{a};
t1.join();
t2.join();

// First thread runs and the initializes counter to 1
// Second thread runs and the initializes counter to 1
// Output: 1 then 1
```

If initialization throws an exception, `std::terminate` is called to shut down the program.

```
struct Throws
{
    Throws()
    {
        throw 123;
    }
};

// Initializing throws an exception which calls
std::terminate
static thread_local Throws t{};
```

Thread-local variables are deallocated and de-initialized when the thread ends:

```
struct LogLifecycle
{
    int Value = 1;

    LogLifecycle()
    {
        DebugLog("ctor");
    }

    ~LogLifecycle()
    {
        DebugLog("dtor");
```

```
    }
};

thread_local LogLifecycle x{};

auto a = []{ DebugLog(x.Value); };
std::thread t1{a};
std::thread t2{a};
t1.join();
t2.join();

// Possible annotated output, depending on thread
execution order:
//   ctor     // first thread initializes x
//   ctor     // second thread initializes x
//   1        // first thread prints x.Value
//   dtor     // first thread de-initializes x
//   1        // second thread prints x.Value
//   dtor     // second thread de-initializes x
```

Any such per-thread initialization and de-initialization needs to be implemented manually in C#.

Volatile

C# and C++ both have a `volatile` keyword, but the meaning is different between the languages. In C#, `volatile` is intended to be used for thread synchronization. It guarantees atomic reads and writes to `volatile` variables, meaning they can't be interrupted by other threads. In order to guarantee atomicity, only certain types can be `volatile` in C#:

- Reference types such as class instances
- Generic type parameters that are reference types such as class instances
- Pointers
- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`
- Enums based on `byte`, `sbyte`, `short`, `ushort`, `int`, and `uint`
- `IntPtr` and `UIntPtr`

All other types, including `double`, `long`, and all structs, can't be `volatile`:

```
// C#
public class Name
{
    public string First;
    public string Last;
}

public struct IntWrapper
{
    public int Value;
```

```
}

public enum IntEnum : int
{
}

public enum LongEnum : long
{
}

unsafe public class Volatiles<T>
    where T : class
{
    // OK: reference type
    volatile Name RefType;

    // OK: type parameter known to be a reference type
due to where constraint
    volatile T TypeParam;

    // OK: pointer
    volatile int* Pointer;

    // OK: permitted primitive type
    volatile int GoodPrimitive;

    // Compiler error: denied primitive type
    volatile long BadPrimitive;
```

```
    // OK: enum based on permitted primitive type
    volatile IntEnum GoodEnum;

    // Compiler error: enum based on denied primitive
type
    volatile LongEnum BadEnum;

    // Compiler error: structs can't be volatile
    // No exception for structs that only have one field
that can be volatile
    volatile IntWrapper Struct;

    // OK: Special-case for IntPtr and UIntPtr structs
    volatile IntPtr SpecialPtr1;
    volatile UIntPtr SpecialPtr2;
}
```

The only variables that can be `volatile` in C# are fields of classes and structs. Local variables and parameters can't be `volatile`.

C# also implicitly adds memory fences to disable instruction reordering and data caching that might be performed by CPUs that execute "out of order." An "acquire fence" is inserted for every read of the `volatile` variable and a "release fence" is inserted for every write:

```
// C#
public struct Counter
```

```
{
    public volatile int Value;

    public void Increment()
    {
        // Reads get an implicit acquire fence
        int cur = this.Value; // acquire-fenced

        int next = cur + 1;

        // Writes get an implicit release fence
        this.Value = next; // release-fenced
    }
}
```

C++, on the other hand, implements `volatile` differently. The keyword is the same, but it's not meant to be used for thread synchronization. Instead, it's meant to implement memory-mapped hardware access:

```
// The hardware device reports its status with a 32-bit
integer
enum class DeviceStatus : int32_t
{
    OK = 0,
    Stuck = 1,
    Fault = 2,
};
```

```
// "Pointer to a volatile DeviceStatus"
// It's stored at a fixed location: the memory-mapped
address
volatile DeviceStatus* pDeviceStatus = (volatile
DeviceStatus*)100;

while (true)
{
    // Read and print the device status
    DebugLog("Device status:", *pDeviceStatus);

    // Wait for one second
    std::this_thread::sleep_for(std::chrono::seconds{1});
}
```

The `volatile` keyword is applied here to the `DeviceStatus` that `pDeviceStatus` points to. This tells the compiler that it cannot assume that it has full visibility into the readers and writers of that 32-bit integer. It has to assume that it may be accessed externally, such as when a device driver writes the device's status to memory address `100`.

As a consequence, the compiler isn't allowed to "optimize" our loop like this:

```
// Only read the pointer once
// Store it as a local variable, likely backed by a
register
```

```
DeviceStatus status = *pDeviceStatus;

while (true)
{
    // Print the device status from the local variable
    // No chance of a cache miss!
    DebugLog("Device status:", status);

    // Wait for one second
    std::this_thread::sleep_for(std::chrono::seconds{1});
}
```

The above “optimization” makes the code faster because there’s no chance of a cache miss when reading through the `pDeviceStatus` pointer. The status is just read once and stored in a CPU register, which is essentially free to read from. The compiler can’t see the kernel driver’s writes to memory address `100`, so it can assume this is a safe optimization.

The only problem is that the device status that we log can no longer change. By marking the value that `pDeviceStatus` points to as `volatile`, the compiler is prohibited from making this optimization. It has to assume that there’s an external writer that might change the device status.

Another effect of `volatile` is that the compiler isn’t allowed to reorder reads and writes to `volatile` variables with respect to other `volatile` variables:

```
// Status from the device
enum class DeviceStatus : int32_t
```

```
{
    OK = 0,
    Stuck = 1,
    Fault = 2,
    CommandAccepted = 3,
    CommandRejected = 4,
};

// Commands to the device
enum class DeviceCommand : int32_t
{
    Retry = 1,
};

// Memory-mapped device I/O
volatile DeviceStatus* pDeviceStatus = (volatile
DeviceStatus*)100;
volatile DeviceCommand* pDeviceCommand = (volatile
DeviceCommand*)200;

while (true)
{
    if (*pDeviceStatus == DeviceStatus::Stuck) // read
    {
        *pDeviceCommand = DeviceCommand::Retry; // write
        while (*pDeviceStatus !=
DeviceStatus::CommandAccepted) // read
        {
```

```
        }
        if (*pDeviceStatus ==
DeviceStatus::CommandRejected || // read
            *pDeviceStatus == DeviceStatus::Stuck) //
read
        {
            throw std::runtime_error{"Failed to get
device un-stuck"};
        }
    }

    // Wait for one second
    std::this_thread::sleep_for(std::chrono::seconds{1});
}
```

Without `volatile`, the compiler would be free to reorder these reads and writes so long as it obeys the “as-if” rule where the code works “as if” the compiler hadn’t done the reordering. Here’s how that might look:

```
// Memory-mapped device I/O without volatile
DeviceStatus* pDeviceStatus = (DeviceStatus*)100;
DeviceCommand* pDeviceCommand = (DeviceCommand*)200;

while (true)
{
    if (*pDeviceStatus == DeviceStatus::Stuck)
    {
```

```
        // Read status first
        DeviceStatus status = *pDeviceStatus;

        // Write command second
        *pDeviceCommand = DeviceCommand::Retry;

        // Check status
        while (status != DeviceStatus::CommandAccepted)
        {
            status = *pDeviceStatus;
        }
        if (*pDeviceStatus ==
DeviceStatus::CommandRejected || // read
            *pDeviceStatus == DeviceStatus::Stuck) //
read
        {
            throw std::runtime_error{"Failed to get
device un-stuck"};
        }
    }

    // Wait for one second
    std::this_thread::sleep_for(std::chrono::seconds{1});
}
```

In this non-`volatile` version, the compiler has decided that we should read the status before we write the command. This might cause us to read and old `CommandRejected` status for a prior command and then throw an exception even when our `Retry`

command was accepted. By applying the `volatile` keyword, we disable such reordering and guarantee that our `volatile` reads and writes occur in the order they're written in.

So far we haven't seen any guarantees from C++ that `volatile` reads and writes are atomic or fenced, as they are in C#. That's because this is simply not the case in C++. This is a critical difference that has implications for how they're used in situations such as multi-threading and for their performance.

Due to this lack of an atomicity guarantee, *any* type may be `volatile` in C++. There's no need to prohibit structs, `double`, and `long` just because accessing them might not be atomic. As we've already seen, pointers (and references) to `volatile` variables can also be taken:

```
struct Vector3d
{
    double X;
    double Y;
    double Z;
};

volatile Vector3d V{2, 4, 6}; // Struct
volatile uint64_t L; // Long
volatile double D; // Double
volatile int A[1000]; // Array
```

Additionally, any variable can be `volatile` in C++. We're not limited to just data members. We can make local variables, nested block variables, parameters, globals, and namespace members `volatile`:

```
volatile int global;

namespace Volatiles
{
    volatile int ns;
}

void Foo(volatile int param)
{
    volatile int local;

    {
        volatile int block;
    }
}
```

The `volatile` keyword is a “type qualifier” like `const`. The shorthands “cv” and “cv-qualified” are commonly used to talk about these two qualifiers. Like `const`, a non-`volatile` type may be implicitly treated as a `volatile` type but not the other way around. The same goes for non-`volatile` `const` types being treated as `const` *and* `volatile` types:

```
int nc_nv = 100;
const int c_nv = 200;
volatile int nc_v = 300;
const volatile int c_v = 400;
```

```
{
    int& i1 = nc_nv; // OK
    int& i2 = c_nv; // Compiler error: removes const
    int& i3 = nc_v; // Compiler error: removes volatile
    int& i4 = c_v; // Compiler error: removes const and
volatile
}

{
    const int& i1 = nc_nv; // OK
    const int& i2 = c_nv; // OK
    const int& i3 = nc_v; // Compiler error: removes
volatile
    const int& i4 = c_v; // Compiler error: removes
volatile
}

{
    volatile int& i1 = nc_nv; // OK
    volatile int& i2 = c_nv; // Compiler error: removes
const
    volatile int& i3 = nc_v; // OK
    volatile int& i4 = c_v; // Compiler error: removes
const
}

{
    const volatile int& i1 = nc_nv; // OK
```

```
    const volatile int& i2 = c_nv; // OK
    const volatile int& i3 = nc_v; // OK
    const volatile int& i4 = c_v; // OK
}
```

The general rule here is that we can treat variables as “more `const`” or “more `volatile`” but not “less `const`” or “less `volatile`” since this would remove important restrictions.

Note that the `mutable` keyword we apply to data members is not a type qualifier like `const`. It is instead a “storage-class-specifier” like `static`, `extern`, or `register`, and `thread_local` that only applies to data members. That’s why we can’t declare a local or global variable with type `mutable int` like we can with `const int`.

Conclusion

Both languages have thread-local storage and a `volatile` keyword, but they have significant differences. Thread-local storage in C++ can be applied to more kinds of variables, such as locals and globals. It also guarantees per-thread initialization where C# only initializes once ever. It also features de-initialization when the thread terminates. C# code needs to manually implement both per-thread initialization and per-thread de-initialization.

As for the `volatile` keyword, it's intended usage and implementation varies significantly between C# and C++. In C#, we get guaranteed atomic accesses and memory fences which is great for synchronizing multiple threads. In C++, we just disable some compiler optimizations that would get in the way of memory-mapped I/O. Thread synchronization is usually solved with other tools, such as mutexes and the Standard Library's `std::atomic` class template. Due to the identical naming of the keyword in both languages, many programmers assume identical functionality. It's important to know that this is *not* the case and to use the keyword appropriately in each language.

33. Alignment, Assembly, and Language Linkage

Alignof

Let's start with a simple operator: `alignof`. We specify a type and it evaluates to a `std::size_t` indicating the type's alignment requirement in terms of number of bytes:

```
struct EmptyStruct
{
};

struct Vector3
{
    float X;
    float Y;
    float Z;
};

// Examples on x64 macOS with Clang compiler
DebugLog(alignof(char)); // 1
DebugLog(alignof(int)); // 4
DebugLog(alignof(bool)); // 1
DebugLog(alignof(int*)); // 8
DebugLog(alignof(EmptyStruct)); // 1
DebugLog(alignof(Vector3)); // 4
DebugLog(alignof(int[100])); // 4
```

Because the alignment requirements of all types in C++ is known at compile time, the `alignof` operator is evaluated at compile time. This means the above is compiled to the same machine code as if we logged constants:

```
DebugLog(1);
DebugLog(4);
DebugLog(1);
DebugLog(8);
DebugLog(1);
DebugLog(4);
DebugLog(4);
```

Note that when using `alignof` with an array, like we did with `int[100]`, we get the alignment of the array's element type. That means we get `4` for `int`, not `8` for `int*` even though arrays and strings are very similar in C++.

Alignas

Next we have the `alignas` specifier. This is applied to classes, data members, and variables to control how they're aligned. All kinds of variables are supported except bit fields, parameters, or variables in catch clauses. For example, say we wanted to align a struct to 16-byte boundaries:

```
// Use default alignment
struct Vector3
{
    float X;
    float Y;
    float Z;
};

// Change alignment to 16 bytes
struct alignas(16) AlignedVector3
{
    float X;
    float Y;
    float Z;
};

DebugLog(alignof(Vector3)); // 4
DebugLog(alignof(AlignedVector3)); // 16
```

We're not allowed to reduce the alignment requirements because the resulting code wouldn't work on the CPU. If we try, we'll get a

compiler error:

```
// Compiler error: requested alignment (1) is lower than
the default (4)
struct alignas(1) AlignedVector3
{
    float X;
    float Y;
    float Z;
};
```

Similarly, invalid alignments also produce a compiler error. What's valid depends on the CPU architecture being compiled for, but usually powers of two are required:

```
// Compiler error: requested alignment (3) is invalid
struct alignas(3) AlignedVector3
{
    float X;
    float Y;
    float Z;
};
```

Aligning to `0` is simply ignored:

```
// OK, but requested alignment (0) is ignored
struct alignas(0) AlignedVector3
{
```

```
    float X;
    float Y;
    float Z;
};

DebugLog(alignof(AlignedVector3)); // 4
```

As a shorthand, we can also use `alignas(type)`. This is equivalent to `alignas(alignof(type))` and it's useful when we want the alignment to match another type's alignment:

```
struct AlignedToDouble
{
    double Double;

    // Each data member has the same alignment as the
double type
    alignas(double) float Float;
    alignas(double) uint16_t Short;
    alignas(double) uint8_t Byte;
};

// Struct is 32 bytes because of alignment requirements
DebugLog(sizeof(AlignedToDouble)); // 32

// Print distances between data members to see 8-byte
alignment
AlignedToDouble atd;
```

```
DebugLog((char*)&atd.Float - (char*)&atd.Double); // 8
DebugLog((char*)&atd.Short - (char*)&atd.Double); // 16
DebugLog((char*)&atd.Byte - (char*)&atd.Double); // 24
```

It's rare, but if we specify multiple `alignas` then the largest value is used:

```
struct Aligned
{
    // 16 is the largest, so it's used as the alignment
    alignas(4) alignas(8) alignas(16) int First = 123;

    alignas(16) int Second = 456;
};

DebugLog(sizeof(Aligned)); // 32

Aligned a;
DebugLog((char*)&a.Second - (char*)&a.First); // 16
```

This leads to the third form of `alignas` where we pass a template parameter pack instead of an integer or a type. In this case, it's just like we specified one `alignas` per element of the parameter pack and therefore the largest value is chosen:

```
template<int... Alignments>
struct Aligned
{
```

```
    alignas(Alignments...) int First = 123;
    alignas(16) int Second = 456;
};

DebugLog(sizeof(Aligned<1, 2, 4, 8, 16>)); // 32

Aligned<1, 2, 4, 8, 16> a;
DebugLog((char*)&a.Second - (char*)&a.First); // 16
```

Assembly

C++ allows us to embed assembly code. This is called “inline assembly” and its meaning is highly-specific to the compiler and the CPU being compiled for. All that the C++ language standard says is that we write `asm("source code")` and the rest is left up to the compiler. For example, here’s some inline assembly that subtracts 5 from 20 on x86 as compiled by Clang on macOS:

```
int difference = 0;
asm(
    "movl $20, %%eax;" // Put 20 in the eax register
    "movl $5, %%ebx;" // Put 5 in the ebx register
    "subl %%ebx, %%eax ":"=a"(difference)); // difference
= eax - ebx
DebugLog(difference); // 15
```

Also compiler-specific is how the assembly code interacts with the surrounding code. In this case, Clang allows us to write `:"=a" (difference)` to reference a the `difference` local variable as an output from inside the `asm` statement.

Each compiler will put its own constraints on inline assembly code. This includes whether the Intel or AT&T assembly syntax is used, how C++ code interacts with the inline assembly, and of course the supported CPU architecture instruction sets.

All of this inconsistency has lead to most uses of inline assembly being eschewed in favor of so-called “intrinsics.” These are functions that are replaced with a single CPU instruction. They are almost always named after that CPU instruction, take the parameters that the CPU instruction operates on, and evaluate to the result of the

CPU instruction. There's a lot of variance in just what this means, but it's a lot simpler and more natural way to embed assembly in a C++ program:

```
// x86 SSE intrinsics
#include <xmmintrin.h>

// Component-wise addition of four floats in two arrays
into a third array
void Add4(const float* a, const float* b, float* c)
{
    // Load a's four floats from memory into a 128-bit
register
    __m128 reg1 = _mm_load_ps(a);

    // Load b's four floats from memory into a 128-bit
register
    const auto reg2 = _mm_load_ps(b);

    // Add corresponding floats of a and b into the first
128-bit register
    reg1 = _mm_add_ps(reg1, reg2);

    // Store the result register into c's memory
    _mm_store_ps(c, reg1);
}

float a[] = { 1, 1, 1, 1 };
float b[] = { 1, 2, 3, 4 };
```

```
float c[] = { 9, 9, 9, 9 };

Add4(a, b, c);

DebugLog(a[0], a[1], a[2], a[3]); // 1, 1, 1, 1
(unmodified)
DebugLog(b[0], b[1], b[2], b[3]); // 1, 2, 3, 4
(unmodified)
DebugLog(c[0], c[1], c[2], c[3]); // 2, 3, 4, 5 (sum)
```

There are several advantages to this approach. Specific register names don't need to be named as the compiler's register allocator simply does its normal work. We're allowed to use normal C++ conventions like parameters, return values, `const` variables, and even `auto` typing. Those variables are strongly-typed, meaning we get the compiler error-checking we're used to:

```
// Compiler error: too many arguments
__m128 reg1 = _mm_load_ps(a, b);

// Compiler error: return value is __m128, not bool
bool reg2 = _mm_load_ps(b);
```

Language Linkage

When object files are linked together by the linker, it's important that they follow the same conventions. Normally this isn't a problem because we're linking together object files compiled from source code in the same language (C++) that was compiled by the same version of the same compiler with the same compiler settings.

In other cases, we want to link together code that was compiled differently. One common scenario is to link together C++ and C code, such as when C++ code is using a C library or visa versa. In this case, the languages have different object file conventions that cause them to clash. Take, for example, the case of overloaded functions in C++. These aren't supported in C, so C's object files simply name the function the same as in the source code. C++ needs to disambiguate, so it "mangles" the names to make them unique. It does this even if there's only one overload of the function:

```
////////////////////
// library.h (C++)
////////////////////

int Madd(int a, int b, int c);

////////////////////
// library.cpp (C++)
////////////////////

#include "library.h"

// Compiled into object file with name Maddiii_i
```

```c
// Example name only. Actual name is effectively
unpredictable.
int Madd(int a, int b, int c)
{
    return a*b + c;
}

////////////////////
// main.c (C)
////////////////////

#include "library.h"

void Foo()
{
    // OK: library.h declares a Madd that takes three
ints and returns an int
    int result = Madd(2, 4, 6);

    // Print the result
    printf("%d\n", result);
}
```

Both `library.cpp` and `main.c` compile, but the linker that takes in `library.o` and `main.o` fails to link them together. The problem is that `main.o` is trying to find a function called `Madd` but there isn't one. There's a function called `Maddiii_i`, but that doesn't count because only exact names are matched.

To solve this problem, C++ provides a way to tell the compiler that code should be compiled with the same language linkage rules as C:

```
////////////////////
// library.h (C++)
////////////////////

// Everything in this block should be compiled with C's
linkage rules
extern "C"
{
    int Madd(int a, int b, int c);
}

////////////////////
// library.cpp (C++)
////////////////////

#include "library.h"

// Definitions need to match the language linkage of
their declarations
extern "C"
{
    // Compiled into object file with name Madd
    // Not mangled into Maddiii_i
    int Madd(int a, int b, int c)
    {
```

```
        return a*b + c;
    }
}
```

Now that `Madd` doesn't have its name mangled the linker can find it and produce a working executable.

Some special rules apply to code that's been switched to C language linkage. First, class members always have C++ linkage regardless of whether C linkage is specified.

Second, because C doesn't support function overloading, any functions with the same name are assumed to be the same function. This means we'll typically get compiler errors for redefining the same function when we try to make an overload.

Third, and similarly, variables in different namespaces with the same name are assumed by the compiler to be the same variable. This is because C doesn't support namespaces. We'll typically get the same compiler errors for trying to redefine these variables.

Fourth, and again similarly, variables and functions can't have the same name even if they're in different namespaces. All of these rules stem from C's requirement that everything has a unique name.

If only a single entity needs its language linkage changed, the curly braces can be omitted similar to how they're optional for one-statement `if` blocks. This doesn't, however, create a block scope as it does with other curly braces:

```
extern "C" int Madd(int a, int b, int c);

extern "C" int Madd(int a, int b, int c)
```

```
{
    return a*b + c;
}
```

The C++ language guarantees that two languages' linkage are supported: C and C++. Compilers are free to implement support for more languages. The default language linkage is clearly C++, but it can be specified explicitly if so desired:

```
extern "C++" int Madd(int a, int b, int c);

extern "C++" int Madd(int a, int b, int c)
{
    return a*b + c;
}
```

Linkage rules can be nested. In this case, the innermost linkage is used:

```
// Change linkage to C
extern "C"
{
    // Change linkage back to C++
    extern "C++" int Madd(int a, int b, int c);
}

// OK: linkage is C++
extern "C++" int Madd(int a, int b, int c)
```

```
{
    return a*b + c;
}
```

Finally, a common convention is to use the preprocessor to check for `__cplusplus` which indicates whether the code is being compiled as C++. In response, C++ language linkage is used. This allows code to be compiled as either C++ as a library that can be linked with C code. The code can also be compiled directly as C, such as when a C++ compiler isn't available. This approach requires the code to use only the subset that is legal for both languages:

```
////////////////////
// library.h
////////////////////

// If compiled as C++, this is defined
#ifdef __cplusplus
    // Make a macro called EXPORTED with the code to set
C language linkage
    #define EXPORTED extern "C"
// If compiled as C (assumed since not C++)
#else
    // Make an empty macro called EXPORTED
    #define EXPORTED
#endif

// Add EXPORTED at the beginning
// For C++, this sets the language linkage to C
```

```
// For C, this does nothing
EXPORTED int Madd(int a, int b, int c);

////////////////////
// library.c
////////////////////

#include "library.h"

// Compiled into object file with name Madd regardless of
language
EXPORTED int Madd(int a, int b, int c)
{
    return a*b + c;
}
```

Conclusion

In addition to the myriad low-level controls C++ gives us, these features provide us with even more control. We can query and set the alignment of various data types and variables to make optimal use of specific CPU architectures' requirements to improve performance in a variety of ways. C# provides some control over struct field layout, but that's a far more limited tool than `alignas` in C++.

One way to use this control over alignment is by writing inline assembly, another C++ feature, or by making use of CPU-specific intrinsics. These features combine together to provide precise control over what machine code actually gets executed and without the need to write entire programs in assembly. C# is beginning to offer intrinsics starting with x86 in .NET Core 3.0 and Unity's Burst-specific intrinsics.

C++ also allows for a high level of compatibility with its predecessor: C. Despite having far more features, C++ code can be easily integrated with C code by setting the language linking mode and following a few special rules. This makes our C++ libraries available for usage in C and in environments that follow C's linkage rules. There are quite a few of those, including language bindings for C#, Rust, Python, and JavaScript via Node.js. The same goes for C# with its P/Invoke system of language bindings that enables interoperability with the C linkage model.

34. Fold Expressions and Elaborated Type Specifiers

Fold Expressions

Fold expressions, available since C++17, allow us to apply a binary operator to all the parameters in a template's parameter pack. For a simple example, say we want to add up some integers:

```
// Template parameters are a pack of integers
template<int... Vals>
// Apply the + operator to the Vals pack
int SumOfAll = (Vals + ...);

// Instantiate the template with four integers in the
Vals pack
DebugLog(SumOfAll<1, 2, 3, 4>); // 10
```

The "fold expression" is the `(Vals + ...)` part. Parentheses are required here, unlike most expressions. We name the parameter pack (`Vals`), name the binary operator (`+`), and add `...` to indicate that we want to fold that operator over the pack.

When the template is instantiated, the compiler converts the fold expression into a series of binary operators:

```
// Expanded version of the template parameter pack
template<int Val1, int Val2, int Val3, int Val4>
```

```
// Expanded version of the fold expression
int SumOfAll = Val1 + (Val2 + (Val3 + Val4));
```

This kind of fold expression is called a “unary right fold.” This means that the rightmost arguments have the operator applied to them fist.

In order to reverse this and apply the operator to the leftmost arguments first, we use a “unary left fold” like this:

```
template<int... Vals>
int SumOfAll = (... + Vals); // Swapped "..." and "Vals"

DebugLog(SumOfAll<1, 2, 3, 4>); // 10
```

When instantiated, the compiler produces the equivalent of this:

```
template<int Val1, int Val2, int Val3, int Val4>
int SumOfAll = ((Val1 + Val2) + Val3) + Val4;
```

The choice of a left or right fold doesn’t really matter when we’re just adding integers, but it will surely matter with other types and other operators.

If the parameter pack happens to be empty, only three binary operators are allowed. First, we can use `&&` to evaluate to `true`:

```
template<bool... Vals>
bool AndAll = (... && Vals);
```

```
DebugLog(AndAll<false, false>); // false
DebugLog(AndAll<false, true>); // false
DebugLog(AndAll<true, false>); // false
DebugLog(AndAll<true, true>); // true
DebugLog(AndAll<>); // true
```

Second, we can use `||` to evaluate to `false`:

```
template<bool... Vals>
bool OrAll = (... || Vals);

DebugLog(OrAll<false, false>); // false
DebugLog(OrAll<false, true>); // true
DebugLog(OrAll<true, false>); // true
DebugLog(OrAll<true, true>); // true
DebugLog(OrAll<>); // false
```

And third, which is by far the least common use case, the `,` operator will evaluate to `void()`:

```
template<bool... Vals>
void Goo()
{
    return (... , Vals); // Equivalent to "return
void();"
}
```

```
// OK
Goo();
```

Now that we've seen the "unary" fold expressions, let's look at the "binary" ones. To make these, we add the same binary operator after the `...` then an additional value:

```
template<int... Vals>
// Add the operator (+) then an additional value (1)
after the unary fold
int SumOfAllPlusOne = (Vals + ... + 1);

DebugLog(SumOfAllPlusOne<1, 2, 3, 4>); // 11
```

Here we've converted the unary fold expression `(Vals + ...)` into a binary one by adding `+ 1` to the end of it. This adds another value in addition to the values in the parameter pack. Since this was a "binary right fold" the parentheses will be added on the rightmost values first:

```
template<int Val1, int Val2, int Val3, int Val4>
int SumOfAll = 1 + (Val1 + (Val2 + (Val3 + Val4)));
```

The "binary left fold" version just has the additional value on the left:

```
template<int... Vals>
int SumOfAllPlusOne = (1 + ... + Vals);
```

When instantiated with four values in the parameter pack, it'll look like this:

```
template<int Val1, int Val2, int Val3, int Val4>
int SumOfAll = (((1 + Val1) + Val2) + Val3) + Val4;
```

Regardless of which kind of fold expression we write, we're allowed to use any of these binary operators:

- `+`
- `-`
- `*`
- `/`
- `%`
- `^`
- `&`
- `|`
- `=`
- `<`
- `>`
- `<<`
- `>>`
- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `^=`

- `&=`
- `|=`
- `<<=`
- `>>=`
- `==`
- `!=`
- `<=`
- `>=`
- `&&`
- `||`
- `,`
- `.*`
- `->*`

Elaborated Type Specifiers

We've seen before that C code requires us to use `struct Player` instead of just `Player` as the type name of the `Player` struct:

```
// C code

struct Player
{
    int Health;
    int Speed;
};

struct Player p; // C requires "struct" prefix
p.Health = 100;
p.Speed = 10;
DebugLog(p.Health, p.Speed); // 100, 10
```

That's *usually* not necessary in C++. However, there is an edge case where we have both a class and a variable with the same name. Using the name refers to the variable, so we're unable to use the type anymore:

```
// A class
struct Player
{
};
```

```
// A variable with the same name as the class
int Player = 123;

// Compiler error: Player is not a type
// This is because "Player" refers to the variable, not the class
Player p;
```

To get around this, we can use the C-style `struct Player` to explicitly state that we're referring to the struct, not the variable. This is called an "elaborated type specifier" since we are elaborating on the `Player` type:

```
// Elaborated type specifier
// OK: refers to the Player struct, not the Player variable
struct Player p;
```

Since `struct` and `class` are very similar, we can use them interchangeably in our elaborated type specifiers:

```
// Elaborated type specifier using "class" when Player is a "struct"
class Player p;
```

Unions are *not* interchangeable:

```
// A union
union IntFloat
{
    int32_t Int;
    float Float;
};

// A variable with the same name as the union
bool IntFloat = true;

// Compiler error: IntFloat is not a type
// This is because "IntFloat" refers to the variable, not
the union
IntFloat u;

// Elaborated type specifier
// Compiler error: IntFloat is a union, not a class or
struct
class IntFloat u;

// Elaborated type specifier
// OK: refers to the IntFloat union, not the IntFloat
variable
union IntFloat u;
```

Enumerations are also their own kind of entity and need to be elaborated with `enum`:

```
// An enumeration
enum DamageType
{
    Physical,
    Water,
    Fire,
    Magic,
};

// A variable with the same name as the enum
float DamageType = 3.14f;

// Elaborated type specifier
// Compiler error: DamageType is an enum, not a class or
struct
class DamageType d;

// Elaborated type specifier
// OK: refers to the DamageType enum, not the DamageType
variable
enum DamageType d;
```

Plain `enum` can be used with a scoped enumeration but `enum class` or `enum struct` can't be used with unscoped enumerations and must be used with scoped enumerations:

```
enum class Scoped
{
```

```
};

enum Unscoped
{
};

enum Scoped e1; // OK
enum Unscoped e2; // OK
enum class Scoped e3; // OK
enum class Unscoped e4; // Compiler error: can't use
scoped with unscoped enum
enum struct Scoped e5; // OK
enum struct Unscoped e6; // Compiler error: can't use
scoped with unscoped enum
```

Regardless of the type, we can also refer to its location in a namespace using the scope resolution operator:

```
namespace Gameplay
{
    enum DamageType
    {
        Physical,
        Water,
        Fire,
        Magic,
    };
```

```
    float DamageType = 3.14f;
}

// Elaborated type specifier using scope resolution
operator
enum Gameplay::DamageType d;
```

The same goes for classes' member types:

```
struct Gameplay
{
    // Member type of the class
    enum DamageType
    {
        Physical,
        Water,
        Fire,
        Magic,
    };

    // Member variable of the class
    constexpr static float DamageType = 3.14f;
};

// Elaborated type specifier referring to class member
type
enum Gameplay::DamageType d;
```

Conclusion

Fold expressions provide a way for us to cleanly apply binary operators to templates' parameter packs. Without them, we'd need to resort to alternatives such as recursively instantiating templates and using specialization to stop the recursion. That's much less readable and much slower to compile as many templates would need to be instantiated and then then thrown away. We get our choice of unary or binary and left or right folds so we can control how the binary operator is applied to the values of the parameter pack. Since C# doesn't have variadic templates, it also doesn't have fold expressions.

Elaborated type specifiers are a minor feature that provides us a workaround in the case where we have types with the same name as variables. We can refer to these types explicitly to change the default meaning of the name shared between the type and the variable. This is rarely necessary, but a nice tool to have when the situation arises. C# doesn't allow a type to have the same name as a variable, so there's no equivalent to this in that language.

35. Modules, The New Build Model

Module Basics

The new build system is based on a new language concept called a "module." This system promises to dramatically decrease compile times, both clean and incremental. It also promises to dramatically increase encapsulation by preventing leakage of preprocessor directives and implementation details. Finally, it fully removes the need to specify file system paths in source code like we do with `#include` and then use complex directory lookups to find the referenced files.

To convert a translation unit such as a `.cpp` file into a "module unit," we use an `export module` statement:

```
///////////
// math.ixx
///////////

export module math;
```

We've done two things here. First, we've named the module with the `.ixx` extension. Module files can be named with any extension, or no extension at all, just like any other C++ source file. The `.ixx` extension is used here simply because it's the preference of Microsoft Visual Studio 2019, one of the first compilers to support modules.

Second, the line `export module math;` begins a module named `math`. Like the rest of C++, the source file is read from top to bottom.

Everything *after* this statement is part of the `math` module, but everything *before* it is not.

Currently the module is empty since there's nothing else in the source file. Let's add some functions:

```
///////////
// math.ixx
///////////

// Normal function before the "export module" statement
float Average(float x, float y)
{
    return (x + y) / 2;
}

// Exported function before the "export module" statement
export float MagnitudeSquared(float x, float y)
{
    return x*x + y*y;
}

// The module begins here
export module math;

// Normal function after the "export module" statement
float Min(float x, float y)
{
    return x < y ? x : y;
```

```
}

// Exported function after the "export module" statement
export float Max(float x, float y)
{
    return x > y ? x : y;
}
```

There are a couple things to notice here, too. First, we can add `export` before anything we want to be usable from outside the module. This includes functions like these, variables, types, using aliases, templates, and namespaces. It does *not* include preprocessor directives such as macros.

Modules can seem analogous to namespaces, but the two are quite distinct. A module can export a namespace and a module doesn't imply a namespace. Modules aren't meant to replace namespaces, but they may be used for similar purposes in grouping together related functionality.

We can export anything that doesn't have internal linkage, such as by being declared `static` or inside an unnamed `namespace`. Our exports must be directly inside of a `namespace` block, outside of any blocks at the top level of the file, or in an `export` block:

```
// Everything in this block is exported
export
{
    float Min(float x, float y)
    {
        return x < y ? x : y;
```

```
    }

    // Redundant "export" has no effect
    export float Max(float x, float y)
    {
        return x > y ? x : y;
    }
}
```

Second, two of these functions are before the `export module math;` statement. These are part of the “global module” rather than the `math` module, just like everything outside of a `namespace` is part of the “global namespace.”

There can be only one module in a module unit source file. This isn’t allowed:

```
// First module: OK
export module math;
float Min(float x, float y)
{
    return x < y ? x : y;
}

// Second module: compiler error
export module util;
export bool IsNearlyZero(float val)
{
```

```
    return val < 0.0001f;
}
```

Assuming we don't do that, let's now use this module from another file:

```
////////////
// main.cpp
////////////

// Import the module for usage
import math;

// OK: Max is found in the "math" module we imported
DebugLog(Max(2, 4)); // 4

// Compiler error: none of these are part of the "math"
module
DebugLog(Average(2, 4));
DebugLog(MagnitudeSquared(2, 4));
DebugLog(Min(2, 4));
```

We use `import` to name the module that we want to use. We get access to everything marked `export` in that module. Unlike with header files, we don't specify the file name of the module unit. This is similar to the C# build system where we simply name a namespace: `using System;`.

Partitions and Fragments

We could put all of the code for a module in a single file, but this doesn't scale well as we add more and more code. Imagine all of `System.Collections.Generic` in a single file! C# addresses this by putting one class (`List<T>`, `Dictionary<K, V>`, etc.) in each file. C++ addresses this in multiple ways. The first is called "module partitions" and they allow us to split code across multiple files while still being part of a single module:

```
///////////////
// geometry.ixx
///////////////

// Specify that this is the "geometry" partition of the
"math" module
export module math:geometry;

export float MagnitudeSquared(float x, float y)
{
    return x * x + y * y;
}

////////////
// stats.ixx
////////////

// Specify that this is the "stats" partition of the
"math" module
```

```
export module math:stats;

export float Min(float x, float y)
{
    return x < y ? x : y;
}

export float Max(float x, float y)
{
    return x > y ? x : y;
}

export float Average(float x, float y)
{
    return (x + y) / 2;
}

////////////
// math.ixx
////////////

// This is the primary "math" module
export module math;

// Import the "stats" partition and export it
export import :stats;

// Import the "geometry" partition and export it
```

```
export import :geometry;

////////////
// main.cpp
////////////

// Import the "math" module as normal
import math;

// Use its exported entities as normal
DebugLog(Min(2, 4)); // 2
DebugLog(Max(2, 4)); // 4
DebugLog(Average(2, 4)); // 3
DebugLog(MagnitudeSquared(2, 4)); // 20
```

We see here that partitions are specified with a `:`. The module partition names the primary module (`math`) and the name of its partition (`stats`). The primary module just uses the name of the partition (`:stats`) because its name (`math`) has already been stated and doesn't need to be repeated. It *must* export all of the partitions so the compiler knows everything that's available in the module when it's used.

Unlike other identifiers, module names may include a `.` in them. This means we could instead use `math.stats` and `math.geometry` as our module names:

```
///////////////
// geometry.ixx
///////////////
```

```
// This is a primary "math.geometry" module
export module math.geometry;

export float MagnitudeSquared(float x, float y)
{
    return x * x + y * y;
}

////////////
// stats.ixx
////////////

// This is a primary "math.stats" module
export module math.stats;

export float Min(float x, float y)
{
    return x < y ? x : y;
}

export float Max(float x, float y)
{
    return x > y ? x : y;
}

export float Average(float x, float y)
{
```

```
    return (x + y) / 2;
}

////////////
// math.ixx
////////////

// This is the primary "math" module
export module math;

// Import the "math.stats" module and export it
export import math.stats;

// Import the "math.geometry" module and export it
export import math.geometry;

////////////
// main.cpp
////////////

// Import the "math" module as normal
import math;

// Use its exported entities as normal
DebugLog(Min(2, 4)); // 2
DebugLog(Max(2, 4)); // 4
DebugLog(Average(2, 4)); // 3
DebugLog(MagnitudeSquared(2, 4)); // 20
```

The difference here is that `math.stats` and `math.geometry` aren't partitions, they're primary modules. Any of them can be used directly:

```
// Import the "math.stats" primary module
import math.stats;

// Use its exported entities as normal
DebugLog(Min(2, 4)); // 2
DebugLog(Max(2, 4)); // 4
DebugLog(Average(2, 4)); // 3
```

It's important to note that `math.stats` and `math.geometry` aren't "submodules" as far as the compiler is concerned. They just happened to be named in a way that makes them appear that way. This is largely the same as C# namespaces since there's no special relationship between `System`, `System.Collections`, and `System.Collections.Generic` other than the naming.

Lastly, there is an implicit `private` "fragment" that can hold *only* code that can't possibly effect the module's interface. This restriction allows compilers to avoid recompiling code that uses the module when only the `private` fragment changes:

```
// Primary module
export module math;

// Export some function declarations
export float Min(float x, float y);
export float Max(float x, float y);
```

```
// This begins the "private fragment"
module :private;

// Define some non-exported functions
float Min(float x, float y)
{
    return x < y ? x : y;
}
float Max(float x, float y)
{
    return x > y ? x : y;
}
```

Module Implementation Units

So far all of our module files have been “module interface units” since they included the `export` keyword. They’re interfaces to be used by code outside the module such as our `main.cpp`.

There’s another kind of module unit though: “module implementation units.” These are meant to contain implementation details of the module. They don’t use the `export` keyword, but contain internal code that’s accessible from within the module:

```
///////////////
// geometry.ixx
///////////////

// A non-exported module partition
module math:geometry;

// A non-exported function
float MagnitudeSquared(float x, float y)
{
    return x * x + y * y;
}

///////////
// math.ixx
///////////

// Primary module
```

```
export module math;

// Import the module implementation partition
import :geometry;

// Export a function from the module implementation
partition by declaring it
// and adding the "export" keyword
export float MagnitudeSquared(float x, float y);

// Export more functions
export float Magnitude(float x, float y)
{
    // Call functions in the imported module
implementation partition
    float magSq = MagnitudeSquared(x, y);
    return Sqrt(magSq); // TODO: write Sqrt()
}
```

This is similar to how we'd split code across header files (`.hpp`) and translation units (`.cpp`). In that traditional build system, we'd add declarations of functions in the header files and definitions of those functions in the translation units.

If we don't need the partitions but still want to separate the interface from the implementation, we can drop the `import` and remove the partition name:

```
///////////////
// geometry.cpp
///////////////

// A non-exported module
module math;

// A non-exported function
float MagnitudeSquared(float x, float y)
{
    return x * x + y * y;
}

////////////
// math.ixx
////////////

export module math;

// Note: no need to "import math;" since this is already
the "math" module

export float MagnitudeSquared(float x, float y);

export float Magnitude(float x, float y)
{
    float magSq = MagnitudeSquared(x, y);
```

```
    return Sqrt(magSq); // TODO: write Sqrt()
}
```

Notice that we now have `geometry.cpp`, not `geometry.ixx`. This is because it can't be imported anymore and must be used implicitly like we did in the `math.ixx` module unit.

Module Linkage

In the traditional build model, there is “internal linkage” and “external linkage.” This means that something is either the same internally in a translation unit or externally across translation units. With modules, there is now “module linkage.” This means that something is the same across all module units and users of the module:

```
//////////////////
// statsglobals.ixx
//////////////////

export module stats:globals;

// Variable with "module linkage"
export int NumEnemiesKilled = 0;

////////////
// stats.ixx
////////////

export module stats;

import :globals;

export void CountEnemyKilled()
{
    // Refers to the same variable as in statsglobal.ixx
    NumEnemiesKilled++;
```

```
}

export int GetNumEnemiesKilled()
{
    // Refers to the same variable as in statsglobal.ixx
    return NumEnemiesKilled;
}

////////////
// main.cpp
////////////

import stats;

DebugLog(GetNumEnemiesKilled()); // 0
CountEnemyKilled();
DebugLog(GetNumEnemiesKilled()); // 1

// Refers to the same variable as in statsglobal.ixx
DebugLog(NumEnemiesKilled); // 1
```

Compatibility

Given the 40+ year history of C++, the new build system must be compatible with the old build system. There are a ton of existing header files that we'll want to use with modules. Thankfully, C++ provides a new preprocessor directive to do just that:

```
import "mylibrary.h";
// ...or...
import <mylibrary.h>;
```

Despite not starting with a `#` and requiring a `;` at the end, this is really a preprocessor directive. It's distinct from a regular module `import` because it either has double quotes (`"mylibrary.h"`) or angle brackets (`<mylibrary.h>`) depending on the header search rules desired.

The effect of this directive is to export everything that's exportable in the header file just like we added `export` to its source code. We typically use it to create a "header unit" that wraps a header file in a module:

```
////////////////
// mylibrary.ixx
////////////////

// Module that wraps mylibrary.h
export module mylibrary;

// Export everything in the header file that can be
```

```
exported
import "mylibrary.h";
```

There are a couple of key differences between this `import` directive and `#include` and `import` with a module. First, contrary to `#include`, preprocessor symbols defined before the `import` directive are not visible to the imported header file:

```
//////////////
// mylibrary.h
//////////////

int ReadVersion()
{
    int version = ReadTextFileAsInteger("version.txt");

    #if ENABLE_LOGGING
        DebugLog("Version: ", version);
    #endif

    return version;
}

///////////
// main.cpp
///////////

#include "mylibrary.h"
```

```
int version = ReadVersion(); // Does not log

// ...equivalent to...

int ReadVersion()
{
    int version = ReadTextFileAsInteger("version.txt");

    #if ENABLE_LOGGING // Note: not defined
        DebugLog("Version: ", version);
    #endif

    return version;
}
int version = ReadVersion();

//////////////////
// mainlogged.cpp
//////////////////

// Define a preprocessor symbol before #include
#define ENABLE_LOGGING 1

#include "mylibrary.h"
int version = ReadVersion(); // Does log

// ...equivalent to...
```

```
#define ENABLE_LOGGING 1

int ReadVersion()
{
    int version = ReadTextFileAsInteger("version.txt");

    #if ENABLE_LOGGING // Note: is defined
        DebugLog("Version: ", version);
    #endif

    return version;
}

int version = ReadVersion();
```

C++ provides a facility to work around this limitation. We can use `module;` before our named module and put preprocessor directives between these two statements. Everything here will be part of the "global module" and accessible from inside the module:

```
///////////////
// metadata.ixx
///////////////

// No module name means "global module"
module;

// Define a preprocessor symbol before #include
```

```
// Only preprocessor symbols are allowed in this section
#define ENABLE_LOGGING 1

// Use #include instead of the import directive
#include "mylibrary.h"

// Our named module
export module metadata;

// Export a function from the header file
export int ReadVersion();

////////////
// main.cpp
////////////

// Use the module as normal
import metadata;
DebugLog(ReadVersion()); // 6
```

The second difference between the `import` directive and `import` with a module is that preprocessor macros in the header file *are* exported:

```
///////////////
// legacymath.h
///////////////
```

```
// Macro defined in the header file
#define PI 3.14

////////////
// math.ixx
////////////

export module math;

// Import directive exposes the PI macro
import "legacymath.h";

export double GetCircumference(double radius)
{
    // Macros from the import directive are usable
    return 2.0 * PI * radius;
}

////////////
// main.cpp
////////////

import math;

// OK
DebugLog(GetCircumference(10.0));

// Compiler error: macros from import directives are not
```

```
exported
DebugLog(PI);
```

Notice how the `PI` macro is available for use in the header unit that used the `import` directive but not in users of that module. This prevents macros from transitively “leaking” throughout an entire program.

Conclusion

C++20's new module build system is much more analogous to C# than its own legacy header files and `#include`. In C++ terms, C# mixes namespaces and modules together somewhat. We write the name of a namespace (`using Math;`) in order to gain access to its contents. C++ separates these two features. We can write `import math;` without `math` being a namespace. We can layer namespaces on top of modules and even `export` them.

C# provides support for splitting code across multiple files by adding one member of a namespace in each file. The same is possible in C++, but we can also go further by adding multiple members in a single file and splitting the interface from the implementation. Partitions and fragments are flexible tools that allow us to sub-divide large modules across many source files.

As a C++20 feature that was only standardized recently, modules are not commonly used as of this writing. However, they're destined to eventually become the dominant build system and bring their many improvements over header files to the vast majority of codebases. In the meantime, we have tools such as the new `import "header.h"` directive and access to the global module to ease the transition. New code using modules can use these tools to package legacy code into modules, just as if it was written that way from the start. Old code can simply continue to use the header files.

36. Coroutines

Fixed Statements

In an `unsafe` context in C#, we can use `fixed` statements to prevent the GC from moving an object:

```
// Unsafe function creates an unsafe context for its body
unsafe void ZeroBytes(byte[] bytes)
{
    // Prevent moving the array
    fixed (byte* pBytes = bytes)
    {
        // Access the array via a pointer
        for (int i = 0; i < bytes.Length; ++i)
        {
            pBytes[i] = 0;
        }
    }
}
```

Since C++ has no GC, our objects never move around. We therefore have no need for a `fixed` statement as we can simply take the address of objects:

```
struct ByteArray
{
    int32_t Length;
```

```
    uint8_t* Bytes;
};

void ZeroBytes(ByteArray& bytes)
{
    ByteArray* pBytes = &bytes;
    for (int i = 0; i < pBytes->Length; ++i)
    {
        pBytes->Bytes[i] = 0;
    }
}
```

There's often no reason to bother with taking a pointer though. This is because objects are passed by value by default. In the above example, we take a `ByteArray&` lvalue reference in `ZeroBytes` because taking just a `ByteArray` would cause a copy to be made when calling the function. So we normally already have a pointer-like reference to objects and can simply use it directly:

```
void ZeroBytes(ByteArray& bytes)
{
    for (int i = 0; i < bytes.Length; ++i)
    {
        bytes.Bytes[i] = 0;
    }
}
```

Fixed Size Buffers

Another meaning of `fixed` in C# is to create a buffer of primitives (`bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`) that is directly part of a `class` or `struct` rather than a managed reference as we'd get with an array such as `byte[]`:

```
// An unsafe context is required
unsafe struct FixedLengthArray
{
    // 16 integers are directly part of the struct
    // This is _not_ a managed reference to an int[]
    public fixed int Elements[16];
}
```

C++ has no need for fixed size buffers as it directly supports arrays:

```
struct FixedLengthArray
{
    // 16 integers are directly part of the struct
    int32_t Elements[16];
};
```

Further, there's no restriction to only use primitive types. Any type may be used:

```
struct Vector2
{
```

```
    float X;
    float Y;
};

struct FixedLengthArray
{
    // 16 Vector2s are directly part of the struct
    Vector2 Elements[16];
};
```

Properties

C# structs and classes support a special kind of function called “properties” that give the illusion that the user is referencing a field rather than calling a function:

```
class Player
{
    // Conventionally called the "backing field"
    string m_Name;

    // Property called Name of type string
    public string Name
    {
        // Its "get" function takes no parameters and
must return the property
        // type: string
        get
        {
            return m_Name;
        }
        // The "set" function is implicitly passed a
single parameter of the
        // property type (string) and must return void
        set
        {
            m_Name = value;
        }
```

```
    }
}

Player p = new Player();

// Call "set" on the Name property and pass "Jackson" as
the value parameter
p.Name = "Jackson";

// Call "get" on the Name property and get the returned
string
DebugLog(p.Name);
```

When the bodies of the `get` and `set` functions and the “backing field” are trivial, as shown above, automatically-implemented properties can be used to tell the compiler to generate this boilerplate:

```
class Player
{
    public string Name { get; set; }
}
```

C++ doesn’t have properties. Instead, naming conventions are typically used to create pairs of “get” and “set” functions. Here’s a popular naming convention:

```
struct Player
{
```

```
    const char* m_Name;

    const char* GetName() const
    {
        return m_Name;
    }

    void SetName(const char* value)
    {
        m_Name = value;
    }
};

Player p{};
p.SetName("Jackson");
DebugLog(p.GetName());
```

Another popular convention relies on overloading to eliminate the “Get” and “Set” prefixes:

```
struct Player
{
    const char* m_Name;

    const char* Name() const
    {
        return m_Name;
    }
```

```
    void Name(const char* value)
    {
        m_Name = value;
    }
};

Player p{};
p.Name("Jackson");
DebugLog(p.Name());
```

Whichever convention is chosen, macros can be used to remove the boilerplate:

```
// Macro to create a property
#define AUTO_PROPERTY(propType, propName) \
    propType m_##propName; \
    const propType& propName() const \
    { \
        return m_##propName; \
    } \
    void propName(const propType& value) \
    { \
        m_##propName = value; \
    }

struct Player
{
```

```
    // Create the property
    AUTO_PROPERTY(const char*, Name)
};

Player p{};
p.Name("Jackson");
DebugLog(p.Name());
```

Extern

To call functions implemented outside of the .NET environment, C# can declare them as `extern`. Typically this is used to call into C or C++ code:

```
using System.Runtime.InteropServices;

public static class WindowsApi
{
    // This function is implemented in Windows'
User32.dll
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(
        IntPtr handle, string message, string caption,
int type);
}

// Call the external function
WindowsApi.MessageBox((IntPtr)0, "Hello!", "Title", 0);
```

The `extern` keyword in C++ has a different meaning: implemented in another translation unit. To call functions in another DLL, we use our platform's API to load the DLL, call functions, then unload it. Here's how we can do that with the Windows API:

```
// Platform API that provides DLL access
#include <windows.h>
```

```
// Load the DLL
auto dll = LoadLibraryA("User32.dll");

// Get the address of the MessageBoxA function (ASCII
version of MessageBox)
auto proc = GetProcAddress(dll, "MessageBoxA");

// Cast to the appropriate kind of function pointer
auto mb = (int32_t(*)(void*, const char*, const char*,
uint32_t))(proc);

// Call MessageBoxA via the function pointer
(*mb)(nullptr, "Hello!", "Title", 0);

// Unload the DLL
FreeLibrary(dll);
```

The .NET environment takes care of loading and unloading DLLs referenced by `[DllImport]` as well as creating function pointers to the associated `extern` functions. As a trade-off, we lose control over elements of the process such as timing and error handling.

For multi-platform C++ code, it's typical to wrap this platform-specific functionality in an abstraction layer that uses the preprocessor to make the right calls. For example:

```
///////////////
// platform.hpp
```

```
///////////////

// Windows
#ifdef _WIN32
    #include <windows.h>
// Non-Windows (e.g. macOS)
#else
    // TODO
#endif

class Platform
{
#if _WIN32
    using MessageBoxFuncPtr = int32_t(*)(
        void*, const char*, const char*, uint32_t);
    HMODULE dll;
    MessageBoxFuncPtr mb;
#else
    // TODO
#endif

public:

    Platform()
    {
#if _WIN32
        dll = LoadLibraryA("User32.dll");
        mb = (MessageBoxFuncPtr)(GetProcAddress(dll,
```

```
"MessageBoxA"));
#else
        // TODO
#endif
    }

    ~Platform()
    {
#if _WIN32
        FreeLibrary(dll);
#else
        // TODO
#endif
    }

    // Abstracts calls to MessageBoxA on Windows and
something else on other
    // platforms (e.g. macOS)
    void MessageBox(const char* message, const char*
title)
    {
#if _WIN32
        (*mb)(nullptr, message, title, 0);
#else
        // TODO
#endif
    }
};
```

```
//////////
// game.cpp
//////////

#include "platform.hpp"

Platform platform{};
platform.MessageBox("Hello!", "Title");
```

One alternative to using preprocessor directives like this is to create different files per platform: `platform_windows.cpp`, `platform_macos.cpp`, etc. Each contains an implementation of the `Platform` class with code appropriate for the platform it's intended to be compiled for. The project can then be configured to only compile one of these files so there will be no link time conflict as only one `Platform` class will exist.

Extension Methods

C# gives the illusion that we can add methods to classes and structs. These are not really added though as they are still `static` functions outside the class or struct. C# just allows for them to be called on instances of the class or struct they “extend”:

```
public static class ArrayExtensions
{
    // Extension method on float[] because the first parameter has "this"
    public static float Average(this float[] array)
    {
        float sum = 0;
        foreach (float cur in array)
        {
            sum += cur;
        }
        return sum / array.Length;
    }
}

float[] array = { 1, 2, 3 };

// Call the extension method like it's a method of float[]
DebugLog(array.Average()); // 2
```

```
// Or call it normally
DebugLog(ArrayExtensions.Average(array));
```

The first version (`array.Average()`) is rewritten by the compiler into the second version (`ArrayExtensions.Average(array)`). Extension methods don't get any special access to the class or struct they contain. For example, they can't access `private` fields.

The C++ version of this is similar to the second version: we typically write a "free function" outside of any class that takes the class to "extend" as a parameter:

```
float Average(float* array, int32_t length)
{
    float sum = 0;
    for (int32_t i = 0; i < length; ++i)
    {
        sum += array[i];
    }
    return sum / length;
}

float array[] = { 1, 2, 3 };
DebugLog(Average(array, 3)); // 2
```

Functions like this could be put into a namespace or made into static member functions of a class, but the principal remains: the function is disconnected from what it "extends" with no special access to it.

Checked Arithmetic

C# features the `checked` keyword to perform runtime checks on arithmetic. We can opt into this on a per-expression basis or for a whole block:

```
public class Player
{
    public uint Health;

    public void TakeDamage(uint amount)
    {
        // Opt into arithmetic checking
        checked
        {
            // If this underflows, an OverflowException
is thrown
            Health -= amount;
        }
    }
}

Player p = new Player{ Health = 100 };

// OK: Health is now 50
p.TakeDamage(50);
```

```
// OverflowException: tried to underflow Health to -20
p.TakeDamage(70);
```

C++ doesn't have built-in arithmetic checking. Instead, we have a few options. First, we can perform our own manual arithmetic checks:

```
struct OverflowException
{
};

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        if (amount > Health)
        {
            throw OverflowException{};
        }
        Health -= amount;
    }
};

Player p{ 100 };

// OK: Health is now 50
```

```
p.TakeDamage(50);

// OverflowException: tried to underflow Health to -20
p.TakeDamage(70);
```

Second, we can wrap numeric types in structs and overload operators with the checks. This option is the closest match to `checked` blocks in C# as it allows us to perform checks on many operations without needing to write anything for each operation:

```
struct CheckedUint32
{
    uint32_t Value;

    // Conversion from uint32_t
    CheckedUint32(uint32_t value)
        : Value(value)
    {
    }

    // Overload the subtraction operator to check for
underflow
    CheckedUint32 operator-(uint32_t amount)
    {
        if (amount > Value)
        {
            throw OverflowException{};
        }
```

```
        return Value - amount;
    }

    // Implicit conversion back to uint32_t
    operator uint32_t()
    {
        return Value;
    }
};

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        // Put Health in a wrapper struct to check its
arithmetic operators
        Health = CheckedUint32{ Health } - amount;
    }
};
```

Or we can create functions that perform checks. This is a close match to `checked` expressions in C# that apply only to one operation:

```
uint32_t CheckedSubtraction(uint32_t a, uint32_t b)
{
    if (b > a)
```

```
    {
        throw OverflowException{};
    }
    return a - b;
}

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        Health = CheckedSubtraction(Health, amount);
    }
};
```

This last approach is taken by libraries such as Boost Checked Arithmetic.

The `unchecked` keyword isn't present in C++ because there's no `checked` arithmetic to disable.

Nameof

C#'s `nameof` operator gets a `string` name of a variable, type, or member:

```
Player p = new Player();
DebugLog(nameof(p)); // p
```

C++ doesn't have this feature built in, but there's a library available that provides a `NAMEOF` macro for similar functionality:

```
Player p{};
DebugLog(NAMEOF(p)); // p
```

As with the C# operator, it supports variables, types, and members. Additionally, it supports macros, enum "flag" values, and operates at both compile time and run time.

Decimal

C# has a built-in `decimal` type for financial calculations and other times where decimal places need to be represented without any rounding:

```
float f = 1.0f;
for (int i = 0; i < 10; ++i)
{
    f -= 0.1f;
    DebugLog(f);
}
```

This prints inaccurate values because floating point can't represent these without rounding:

```
0.9
0.8
0.6999999
0.5999999
0.4999999
0.3999999
0.2999999
0.1999999
0.09999993
-7.450581E-08
```

If we use `decimal`, we avoid the rounding:

```
decimal d = 1.0m;
for (int i = 0; i < 10; ++i)
{
    d -= 0.1m;
    DebugLog(d);
}
```

This prints:

```
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
```

C++ doesn't have a built-in `decimal` type, but libraries such as GMP and decimal_for_cpp create such types. For example, in the latter library we can write this:

```
#include "decimal.h"
using namespace dec;
```

```
decimal<1> d{ 1.0 };
for (int i = 0; i < 10; ++i)
{
    d -= decimal<1>{ 0.1 };
    DebugLog(d);
}
```

This prints what we'd expect:

```
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
```

Reflection

C# implicitly stores a lot of information about the structure of the program in the binaries it compiles to. This information is then accessible at runtime for the C# code to query via “reflection” methods like `GetType` that return classes like `Type`.

```
public class Player
{
    public string Name;
    public uint Health;
}

Player p = new Player{Name="Jackson", Health=100};
Type type = p.GetType();
foreach (FieldInfo fi in type.GetFields())
{
    DebugLog(fi.Name + ": " + fi.GetValue(p));
}
```

This prints:

```
Name: Jackson
Health: 100
```

The only information like this that C++ stores is data for RTTI to support `dynamic_cast` and `typeid`. It’s a very small subset of what’s available in C# since even full type names are not usually preserved

in `typeid` and only classes with virtual functions are supported by `dynamic_cast`.

So if we want to store this information, we need to store it ourselves. We could do this manually by implementing our own reflection system:

```
// Different types our reflection system supports
enum class Type
{
    None,
    ConstCharPointer,
    Uint32,
};

// Reflected values
struct Value
{
    // Type of the value
    Type Type;

    // Pointer to the value
    void* ValuePtr;
};

// "Interface" to "implement" to make a class support
reflection
struct IReflectable
{
```

```
    using MemberName = const char*;

    // Get names of the class' fields
    virtual const MemberName* GetFieldNames() = 0;

    // Get a value of a class instance's field
    virtual Value GetFieldValue(MemberName* name) = 0;
};

// Player supports reflection
class Player : IReflectable
{
    // Names of the fields. Initialized after the class.
    static const char* const FieldNames[3];

public:

    const char* Name;
    uint32_t Health;

    virtual const MemberName* GetFieldNames() override
    {
        return FieldNames;
    }

    virtual Value GetFieldValue(MemberName* name)
override
    {
```

```
        // strcmp is a Standard Library function
returning 0 when strings equal
        if (!strcmp(name, "Name"))
        {
            return { Type::ConstCharPointer, &Name };
        }
        else if (!strcmp(name, "Health"))
        {
            return { Type::Uint32, &Health };
        }
        return { Type::None, nullptr };
    }
};
const char* const Player::FieldNames[3]{ "Name",
"Health", nullptr };

Player p;
p.Name = "Jackson";
p.Health = 100;

auto fieldNames = p.GetFieldNames();
for (int32_t i = 0; fieldNames[i]; ++i)
{
    auto fieldName = fieldNames[i];
    auto fieldValue = p.GetFieldValue(fieldName);
    switch (fieldValue.Type)
    {
    case Type::ConstCharPointer:
```

```
            DebugLog(fieldName, ": ", *(const
char**)fieldValue.ValuePtr);
            break;
        case Type::Uint32:
            DebugLog(fieldName, ": ", *
(uint32_t*)fieldValue.ValuePtr);
            break;
        }
}
```

This prints the same logs:

```
Name: Jackson
Health: 100
```

Manually adding all of this is quite tedious and creates a maintenance problem as the code changes. As a result, there are many reflection libraries available for C++ to remove a lot of the boilerplate:

- Boost PFR provides basic reflection
- Magic Enum supports only enums
- RTTR has more complete reflection features

For example, in RTTR we can write just this:

```
#include <rttr/registration>
using namespace rttr;
```

```
class Player
{
    const char* Name;
    uint32_t Health;
};

RTTR_REGISTRATION
{
    registration::class_<Player>("Player")
        .property("Name", &Player::Name)
        .property("Health", &Player::Health);
}

Player p;
p.Name = "Jackson";
p.Health = 100;

type t = type::get<Player>();
for (auto& prop : t.get_properties())
{
    DebugLog(prop.get_name(), ": ", prop.get_value(p));
}
```

Conclusion

Neither language is a subset of the other. In almost every chapter of this book, we've seen how the C++ version of various language features is larger and more powerful than the C# equivalent. In this chapter we've seen the opposite: several features that C# has that C++ doesn't.

We've also seen how to at least approximate that functionality in C++ when it's desired. Sometimes, as in the case of `fixed` statements and buffers, there's no need for such a feature in C++ and we can simply stop using the C# feature.

Other times, as with extension methods and properties, there's no direct equivalent and we'll need to tweak our design to fit C++ norms such as the use of free functions and "GetX" functions.

Then there are some cases where libraries are available to implement similar functionality on top of the C++ language. This is the case with `decimal`, `nameof`, and reflection. The powerful, relatively low-level tools that C++ provides makes the effecient implementation of such libraries possible.

Finally, there are some missing C# features whose alternatives depend on the Standard Library specifically. We'll see those alternatives later on in the book.

37. Missing Language Features

Fixed Statements

In an `unsafe` context in C#, we can use `fixed` statements to prevent the GC from moving an object:

```
// Unsafe function creates an unsafe context for its body
unsafe void ZeroBytes(byte[] bytes)
{
    // Prevent moving the array
    fixed (byte* pBytes = bytes)
    {
        // Access the array via a pointer
        for (int i = 0; i < bytes.Length; ++i)
        {
            pBytes[i] = 0;
        }
    }
}
```

Since C++ has no GC, our objects never move around. We therefore have no need for a `fixed` statement as we can simply take the address of objects:

```
struct ByteArray
{
    int32_t Length;
```

```
    uint8_t* Bytes;
};

void ZeroBytes(ByteArray& bytes)
{
    ByteArray* pBytes = &bytes;
    for (int i = 0; i < pBytes->Length; ++i)
    {
        pBytes->Bytes[i] = 0;
    }
}
```

There's often no reason to bother with taking a pointer though. This is because objects are passed by value by default. In the above example, we take a `ByteArray&` lvalue reference in `ZeroBytes` because taking just a `ByteArray` would cause a copy to be made when calling the function. So we normally already have a pointer-like reference to objects and can simply use it directly:

```
void ZeroBytes(ByteArray& bytes)
{
    for (int i = 0; i < bytes.Length; ++i)
    {
        bytes.Bytes[i] = 0;
    }
}
```

Fixed Size Buffers

Another meaning of `fixed` in C# is to create a buffer of primitives (`bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`) that is directly part of a `class` or `struct` rather than a managed reference as we'd get with an array such as `byte[]`:

```
// An unsafe context is required
unsafe struct FixedLengthArray
{
    // 16 integers are directly part of the struct
    // This is _not_ a managed reference to an int[]
    public fixed int Elements[16];
}
```

C++ has no need for fixed size buffers as it directly supports arrays:

```
struct FixedLengthArray
{
    // 16 integers are directly part of the struct
    int32_t Elements[16];
};
```

Further, there's no restriction to only use primitive types. Any type may be used:

```
struct Vector2
{
```

```
    float X;
    float Y;
};

struct FixedLengthArray
{
    // 16 Vector2s are directly part of the struct
    Vector2 Elements[16];
};
```

Properties

C# structs and classes support a special kind of function called “properties” that give the illusion that the user is referencing a field rather than calling a function:

```
class Player
{
    // Conventionally called the "backing field"
    string m_Name;

    // Property called Name of type string
    public string Name
    {
        // Its "get" function takes no parameters and
must return the property
        // type: string
        get
        {
            return m_Name;
        }
        // The "set" function is implicitly passed a
single parameter of the
        // property type (string) and must return void
        set
        {
            m_Name = value;
        }
```

```
    }
}

Player p = new Player();

// Call "set" on the Name property and pass "Jackson" as
the value parameter
p.Name = "Jackson";

// Call "get" on the Name property and get the returned
string
DebugLog(p.Name);
```

When the bodies of the `get` and `set` functions and the “backing field” are trivial, as shown above, automatically-implemented properties can be used to tell the compiler to generate this boilerplate:

```
class Player
{
    public string Name { get; set; }
}
```

C++ doesn’t have properties. Instead, naming conventions are typically used to create pairs of “get” and “set” functions. Here’s a popular naming convention:

```
struct Player
{
```

```
    const char* m_Name;

    const char* GetName() const
    {
        return m_Name;
    }

    void SetName(const char* value)
    {
        m_Name = value;
    }
};

Player p{};
p.SetName("Jackson");
DebugLog(p.GetName());
```

Another popular convention relies on overloading to eliminate the “Get” and “Set” prefixes:

```
struct Player
{
    const char* m_Name;

    const char* Name() const
    {
        return m_Name;
    }
```

```
    void Name(const char* value)
    {
        m_Name = value;
    }
};

Player p{};
p.Name("Jackson");
DebugLog(p.Name());
```

Whichever convention is chosen, macros can be used to remove the boilerplate:

```
// Macro to create a property
#define AUTO_PROPERTY(propType, propName) \
    propType m_##propName; \
    const propType& propName() const \
    { \
        return m_##propName; \
    } \
    void propName(const propType& value) \
    { \
        m_##propName = value; \
    }

struct Player
{
```

```
    // Create the property
    AUTO_PROPERTY(const char*, Name)
};

Player p{};
p.Name("Jackson");
DebugLog(p.Name());
```

Extern

To call functions implemented outside of the .NET environment, C# can declare them as `extern`. Typically this is used to call into C or C++ code:

```
using System.Runtime.InteropServices;

public static class WindowsApi
{
    // This function is implemented in Windows'
User32.dll
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(
        IntPtr handle, string message, string caption,
int type);
}

// Call the external function
WindowsApi.MessageBox((IntPtr)0, "Hello!", "Title", 0);
```

The `extern` keyword in C++ has a different meaning: implemented in another translation unit. To call functions in another DLL, we use our platform's API to load the DLL, call functions, then unload it. Here's how we can do that with the Windows API:

```
// Platform API that provides DLL access
#include <windows.h>
```

```
// Load the DLL
auto dll = LoadLibraryA("User32.dll");

// Get the address of the MessageBoxA function (ASCII
version of MessageBox)
auto proc = GetProcAddress(dll, "MessageBoxA");

// Cast to the appropriate kind of function pointer
auto mb = (int32_t(*)(void*, const char*, const char*,
uint32_t))(proc);

// Call MessageBoxA via the function pointer
(*mb)(nullptr, "Hello!", "Title", 0);

// Unload the DLL
FreeLibrary(dll);
```

The .NET environment takes care of loading and unloading DLLs referenced by `[DllImport]` as well as creating function pointers to the associated `extern` functions. As a trade-off, we lose control over elements of the process such as timing and error handling.

For multi-platform C++ code, it's typical to wrap this platform-specific functionality in an abstraction layer that uses the preprocessor to make the right calls. For example:

```
///////////////
// platform.hpp
```

```
///////////////

// Windows
#ifdef _WIN32
    #include <windows.h>
// Non-Windows (e.g. macOS)
#else
    // TODO
#endif

class Platform
{
#if _WIN32
    using MessageBoxFuncPtr = int32_t(*)(
        void*, const char*, const char*, uint32_t);
    HMODULE dll;
    MessageBoxFuncPtr mb;
#else
    // TODO
#endif

public:

    Platform()
    {
#if _WIN32
        dll = LoadLibraryA("User32.dll");
        mb = (MessageBoxFuncPtr)(GetProcAddress(dll,
```

```
"MessageBoxA"));
#else
        // TODO
#endif
    }

    ~Platform()
    {
#if _WIN32
        FreeLibrary(dll);
#else
        // TODO
#endif
    }

    // Abstracts calls to MessageBoxA on Windows and
something else on other
    // platforms (e.g. macOS)
    void MessageBox(const char* message, const char*
title)
    {
#if _WIN32
        (*mb)(nullptr, message, title, 0);
#else
        // TODO
#endif
    }
};
```

```
//////////
// game.cpp
//////////

#include "platform.hpp"

Platform platform{};
platform.MessageBox("Hello!", "Title");
```

One alternative to using preprocessor directives like this is to create different files per platform: `platform_windows.cpp`, `platform_macos.cpp`, etc. Each contains an implementation of the `Platform` class with code appropriate for the platform it's intended to be compiled for. The project can then be configured to only compile one of these files so there will be no link time conflict as only one `Platform` class will exist.

Extension Methods

C# gives the illusion that we can add methods to classes and structs. These are not really added though as they are still `static` functions outside the class or struct. C# just allows for them to be called on instances of the class or struct they “extend”:

```
public static class ArrayExtensions
{
    // Extension method on float[] because the first parameter has "this"
    public static float Average(this float[] array)
    {
        float sum = 0;
        foreach (float cur in array)
        {
            sum += cur;
        }
        return sum / array.Length;
    }
}

float[] array = { 1, 2, 3 };

// Call the extension method like it's a method of float[]
DebugLog(array.Average()); // 2
```

```
// Or call it normally
DebugLog(ArrayExtensions.Average(array));
```

The first version (`array.Average()`) is rewritten by the compiler into the second version (`ArrayExtensions.Average(array)`). Extension methods don't get any special access to the class or struct they contain. For example, they can't access `private` fields.

The C++ version of this is similar to the second version: we typically write a "free function" outside of any class that takes the class to "extend" as a parameter:

```
float Average(float* array, int32_t length)
{
    float sum = 0;
    for (int32_t i = 0; i < length; ++i)
    {
        sum += array[i];
    }
    return sum / length;
}

float array[] = { 1, 2, 3 };
DebugLog(Average(array, 3)); // 2
```

Functions like this could be put into a namespace or made into static member functions of a class, but the principal remains: the function is disconnected from what it "extends" with no special access to it.

Checked Arithmetic

C# features the `checked` keyword to perform runtime checks on arithmetic. We can opt into this on a per-expression basis or for a whole block:

```
public class Player
{
    public uint Health;

    public void TakeDamage(uint amount)
    {
        // Opt into arithmetic checking
        checked
        {
            // If this underflows, an OverflowException
is thrown
            Health -= amount;
        }
    }
}

Player p = new Player{ Health = 100 };

// OK: Health is now 50
p.TakeDamage(50);
```

```
// OverflowException: tried to underflow Health to -20
p.TakeDamage(70);
```

C++ doesn't have built-in arithmetic checking. Instead, we have a few options. First, we can perform our own manual arithmetic checks:

```
struct OverflowException
{
};

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        if (amount > Health)
        {
            throw OverflowException{};
        }
        Health -= amount;
    }
};

Player p{ 100 };

// OK: Health is now 50
```

```
p.TakeDamage(50);

// OverflowException: tried to underflow Health to -20
p.TakeDamage(70);
```

Second, we can wrap numeric types in structs and overload operators with the checks. This option is the closest match to `checked` blocks in C# as it allows us to perform checks on many operations without needing to write anything for each operation:

```
struct CheckedUint32
{
    uint32_t Value;

    // Conversion from uint32_t
    CheckedUint32(uint32_t value)
        : Value(value)
    {
    }

    // Overload the subtraction operator to check for
underflow
    CheckedUint32 operator-(uint32_t amount)
    {
        if (amount > Value)
        {
            throw OverflowException{};
        }
```

```
        return Value - amount;
    }

    // Implicit conversion back to uint32_t
    operator uint32_t()
    {
        return Value;
    }
};

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        // Put Health in a wrapper struct to check its
arithmetic operators
        Health = CheckedUint32{ Health } - amount;
    }
};
```

Or we can create functions that perform checks. This is a close match to `checked` expressions in C# that apply only to one operation:

```
uint32_t CheckedSubtraction(uint32_t a, uint32_t b)
{
    if (b > a)
```

```
    {
        throw OverflowException{};
    }
    return a - b;
}

struct Player
{
    uint32_t Health;

    void TakeDamage(uint32_t amount)
    {
        Health = CheckedSubtraction(Health, amount);
    }
};
```

This last approach is taken by libraries such as Boost Checked Arithmetic.

The `unchecked` keyword isn't present in C++ because there's no `checked` arithmetic to disable.

Nameof

C#'s `nameof` operator gets a `string` name of a variable, type, or member:

```
Player p = new Player();
DebugLog(nameof(p)); // p
```

C++ doesn't have this feature built in, but there's a library available that provides a `NAMEOF` macro for similar functionality:

```
Player p{};
DebugLog(NAMEOF(p)); // p
```

As with the C# operator, it supports variables, types, and members. Additionally, it supports macros, enum "flag" values, and operates at both compile time and run time.

Decimal

C# has a built-in `decimal` type for financial calculations and other times where decimal places need to be represented without any rounding:

```
float f = 1.0f;
for (int i = 0; i < 10; ++i)
{
    f -= 0.1f;
    DebugLog(f);
}
```

This prints inaccurate values because floating point can't represent these without rounding:

```
0.9
0.8
0.6999999
0.5999999
0.4999999
0.3999999
0.2999999
0.1999999
0.09999993
-7.450581E-08
```

If we use `decimal`, we avoid the rounding:

```
decimal d = 1.0m;
for (int i = 0; i < 10; ++i)
{
    d -= 0.1m;
    DebugLog(d);
}
```

This prints:

```
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
```

C++ doesn't have a built-in `decimal` type, but libraries such as GMP and decimal_for_cpp create such types. For example, in the latter library we can write this:

```
#include "decimal.h"
using namespace dec;
```

```
decimal<1> d{ 1.0 };
for (int i = 0; i < 10; ++i)
{
    d -= decimal<1>{ 0.1 };
    DebugLog(d);
}
```

This prints what we'd expect:

```
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
```

Reflection

C# implicitly stores a lot of information about the structure of the program in the binaries it compiles to. This information is then accessible at runtime for the C# code to query via “reflection” methods like `GetType` that return classes like `Type`.

```
public class Player
{
    public string Name;
    public uint Health;
}

Player p = new Player{Name="Jackson", Health=100};
Type type = p.GetType();
foreach (FieldInfo fi in type.GetFields())
{
    DebugLog(fi.Name + ": " + fi.GetValue(p));
}
```

This prints:

```
Name: Jackson
Health: 100
```

The only information like this that C++ stores is data for RTTI to support `dynamic_cast` and `typeid`. It’s a very small subset of what’s available in C# since even full type names are not usually preserved

in `typeid` and only classes with virtual functions are supported by `dynamic_cast`.

So if we want to store this information, we need to store it ourselves. We could do this manually by implementing our own reflection system:

```
// Different types our reflection system supports
enum class Type
{
    None,
    ConstCharPointer,
    Uint32,
};

// Reflected values
struct Value
{
    // Type of the value
    Type Type;

    // Pointer to the value
    void* ValuePtr;
};

// "Interface" to "implement" to make a class support
reflection
struct IReflectable
{
```

```
    using MemberName = const char*;

    // Get names of the class' fields
    virtual const MemberName* GetFieldNames() = 0;

    // Get a value of a class instance's field
    virtual Value GetFieldValue(MemberName* name) = 0;
};

// Player supports reflection
class Player : IReflectable
{
    // Names of the fields. Initialized after the class.
    static const char* const FieldNames[3];

public:

    const char* Name;
    uint32_t Health;

    virtual const MemberName* GetFieldNames() override
    {
        return FieldNames;
    }

    virtual Value GetFieldValue(MemberName* name)
override
    {
```

```
        // strcmp is a Standard Library function
returning 0 when strings equal
        if (!strcmp(name, "Name"))
        {
            return { Type::ConstCharPointer, &Name };
        }
        else if (!strcmp(name, "Health"))
        {
            return { Type::Uint32, &Health };
        }
        return { Type::None, nullptr };
    }
};
const char* const Player::FieldNames[3]{ "Name",
"Health", nullptr };

Player p;
p.Name = "Jackson";
p.Health = 100;

auto fieldNames = p.GetFieldNames();
for (int32_t i = 0; fieldNames[i]; ++i)
{
    auto fieldName = fieldNames[i];
    auto fieldValue = p.GetFieldValue(fieldName);
    switch (fieldValue.Type)
    {
    case Type::ConstCharPointer:
```

```
            DebugLog(fieldName, ": ", *(const
char**)fieldValue.ValuePtr);
            break;
        case Type::Uint32:
            DebugLog(fieldName, ": ", *
(uint32_t*)fieldValue.ValuePtr);
            break;
        }
}
```

This prints the same logs:

```
Name: Jackson
Health: 100
```

Manually adding all of this is quite tedious and creates a maintenance problem as the code changes. As a result, there are many reflection libraries available for C++ to remove a lot of the boilerplate:

- Boost PFR provides basic reflection
- Magic Enum supports only enums
- RTTR has more complete reflection features

For example, in RTTR we can write just this:

```
#include <rttr/registration>
using namespace rttr;
```

```
class Player
{
    const char* Name;
    uint32_t Health;
};

RTTR_REGISTRATION
{
    registration::class_<Player>("Player")
        .property("Name", &Player::Name)
        .property("Health", &Player::Health);
}

Player p;
p.Name = "Jackson";
p.Health = 100;

type t = type::get<Player>();
for (auto& prop : t.get_properties())
{
    DebugLog(prop.get_name(), ": ", prop.get_value(p));
}
```

Conclusion

Neither language is a subset of the other. In almost every chapter of this book, we've seen how the C++ version of various language features is larger and more powerful than the C# equivalent. In this chapter we've seen the opposite: several features that C# has that C++ doesn't.

We've also seen how to at least approximate that functionality in C++ when it's desired. Sometimes, as in the case of `fixed` statements and buffers, there's no need for such a feature in C++ and we can simply stop using the C# feature.

Other times, as with extension methods and properties, there's no direct equivalent and we'll need to tweak our design to fit C++ norms such as the use of free functions and "GetX" functions.

Then there are some cases where libraries are available to implement similar functionality on top of the C++ language. This is the case with `decimal`, `nameof`, and reflection. The powerful, relatively low-level tools that C++ provides makes the effecient implementation of such libraries possible.

Finally, there are some missing C# features whose alternatives depend on the Standard Library specifically. We'll see those alternatives later on in the book.

38. C Standard Library

Background

First, a word of caution: the C Standard Library is very old. Most of it dates back at least 30 years and even the newer parts are about 10 years old and built to fit in with the original design. The C language itself is also very simple. Its lack of features impacts the library design.

For example, there are "families" of functions that all do the same thing but on different data types. To take an absolute value of a floating point value we call `fabs` for `double`, `fabsf` for `float`, and `fabsl` for `long double`. In C++, we'd just overload `abs` with different parameter types and the compiler would choose the right one to call.

The C++ Standard Library includes many more modern designs that rely on C++ language features. It has that `abs` overloaded function, for example. The C Standard Library is included in the C++ Standard Library largely as part of C++'s broad goal to maintain a high degree of compatibility with C code. There are a few parts of it that are genuinely useful on their own, but these are few and far between.

Still, 30+ years of momentum is a powerful force and it's extremely common to see the C Standard Library in use even when more modern alternatives are available. That makes it important for us to understand as many C++ codebases will include some C Standard Library usage.

We're not going to go in depth and cover every little corner of the C Standard Library in this chapter, but we'll survey its highlights.

General Purpose

As for composition, the C++ Standard Library is made up of header files. As of C++20, modules are also available. The C Standard Library is available only as header files. C Standard Library header files are named with a `.h` extension: `math.h`. These can be included directly into C++ files: `#include <math.h>`. They are also wrapped by the C++ Standard Library. The wrapped versions begin with a `c` and drop the `.h` extension, so we can `#include <cmath>`. These wrapped header files place everything in the `std` namespace and *may* also place everything in the global namespace so both `std::fabs` and `::fabs` work.

There's one truly general purpose header file in the C Standard Library: `stdlib.h`/`cstdlib`. Unlike a more focused header file like `math.h`/`cmath` that obviously focuses on mathematics, a variety of utilities are provided by this header. Some of the basics include `size_t`, the type that the `sizeof` operator evaluates to, and `NULL`, a null pointer constant widely used before the advent of `nullptr` in C++11. The broad nature of this header file makes it hard to compare to C#, but it can roughly be though of as the `System` namespace:

```
#include <stdlib.h>

// sizeof() evaluates to size_t
size_t intSize = sizeof(int);
DebugLog(intSize); // Maybe 4

// NULL can be used as a pointer to indicate "null"
int* ptr = NULL;
```

```
// It's vulnerable to accidental misuse in arithemtic
int sum = NULL + NULL;

// nullptr isn't: this is a compiler error
int sum2 = nullptr + nullptr;
```

Before C++ introduced the `new` and `delete` operators for dynamic memory allocation, C code would use the `malloc`, `calloc`, `realloc`, and `free` functions. The C# equivalent of `malloc` is `Marshal.AllocHGlobal`, `realloc` is `Marshal.ReallocHGlobal`, and `free` is `Marshal.FreeHGlobal`:

```
// Allocate 1 KB of uninitialized memory
// Returns null upon failure
// Memory is untyped, so casting is required to read or
write
void* memory = malloc(1024);

// Reading it before initialization is undefined behavior
int firstInt = ((int*)memory)[0];

// Release the memory. Failing to do so is a memory leak.
free(memory);

// Allocate and initialize to all zeroes 1 KB of memory:
256 x 4 bytes
memory = calloc(256, 4);
```

```
// Re-allocate previously-allocated memory to get more or
less
// Old memory is not freed if allocation fails
memory = realloc(memory, 2048);

// Also need to release memory from calloc and realloc
free(memory);
```

There are some functions to parse numbers from strings, similar to `int.Parse`, `float.Parse`, etc.:

```
// Parse a double
double d = atof("3.14");
DebugLog(d); // 3.14

// Parse an int
int i = atoi("123");
DebugLog(i); // 123

// Parse a float and get a pointer to its end in a string
const char* floatStr = "2.2 123.456";
char* pEnd;
float f = strtof(floatStr, &pEnd);
DebugLog(f); // 2.2

// Use the end pointer to parse more
```

```
f = strtof(pEnd, &pEnd);
DebugLog(f); // 123.456
```

Some generic algorithms are provided, similar to the C# `Array` class as well as `Random` and `Math`:

```
// Seed the global randomizer
// This is not thread-safe
srand(123);

// Use the global randomizer to generate a random number
int r = rand();
DebugLog(r); // Maybe 440

// Compare pointers to ints
auto compare = [](const void* a, const void* b) {
    return *(int*)a - *(int*)b;
};

// Sort an array
int a[] = { 4, 2, 1, 3 };
qsort(a, 4, sizeof(int), compare);
DebugLog(a[0], a[1], a[2], a[3]); // 1, 2, 3, 4

// Binary search the array for 2
int valToFind = 2;
int* pVal = (int*)bsearch(&valToFind, a, 4, sizeof(int),
compare);
```

```
int index = pVal - a;
DebugLog(index); // 1

// Take an absolute value
DebugLog(abs(-10)); // 10

// Divide and also get the remainder
// stdlib.h/cstdlib also provides the div_t struct type
div_t d = div(11, 3);
DebugLog(d.quot, d.rem); // 3, 2
```

Finally, there's some OS-related functionality:

```
// Run a system command
int exitCode = system("ping example.com");
DebugLog(exitCode); // 0 if successful

// Get an environment variable
char* path = getenv("PATH");
DebugLog(path); // Path to executables

// Register a function (lambda in this case) to be called
when the program exits
atexit([]{ DebugLog("Exiting..."); });

// Explicitly exit the program with an exit code
exit(1); // Exiting...
```

Math and Numbers

The next category of header in the C Standard Library relates to mathematics. One we've seen throughout the book is `stdint.h`/`cstdint`, which provides integer types via typedef. Basic types like `int` have guaranteed sizes in C#, but this header file goes above and beyond to also define types that fulfill particular requirements:

```
#include <stdint.h>

int32_t i32; // Always signed 32-bit
int_fast32_t if32; // Fastest signed integer type with at
least 32 bits
intptr_t ip; // Signed integer that can hold a pointer
int_least32_t il; // Smallest signed integer with at
least 32 bits
intmax_t imax; // Biggest available signed integer

// Range of 32-bit integer values
DebugLog(INT32_MIN, INT32_MAX); // -2147483648,
2147483647

// Biggest size_t
DebugLog(SIZE_MAX); // Maybe 18446744073709551615
```

There are also some types in `stddef.h`/`cstddef`. Some of these are more types that satisfy particular requirements. Unusually, there are

also types that are C++-specific in the `cstddef` version of this header:

```
#include <cstddef>

// C and C++ types
std::max_align_t ma; // Type with the biggest alignment
std::ptrdiff_t pd; // Big enough to hold the subtraction
of two pointers

// C++-specific types
std::nullptr_t np = nullptr; // The type of nullptr
std::byte b; // An "enum class" version of a single byte
```

`limits.h`/`climits` also has some maximum and minimum macros, equivalent to `int.MaxValue` and similar in C#:

```
#include <limits.h>

// Range of int values
DebugLog(INT_MIN, INT_MAX); // Maybe -2147483648,
2147483647

// Range of char values
DebugLog(CHAR_MIN, CHAR_MAX); // Maybe -128, 127
```

The `inttypes.h`/`cinttypes` header also has integer-related utilities. These are needed because conversions to and from strings aren't

built into the language as they are in C# with functions like `int.Parse`:

```
#include <inttypes.h>

// Parse a hexadecimal string to an int
// The nullptr means we don't want to get a pointer to
the end
intmax_t i = strtoimax("f0a2", nullptr, 16);
DebugLog(i); // 61602
```

Similarly, `float.h`/`cfloat` provides a bunch of floating point macros similar to what C# provides via constants like `float.MaxValue`:

```
#include <float.h>

// Biggest float
float f = FLT_MAX;
DebugLog(f); // 3.40282e+38

// Difference between 1.0 and the next larger float
float ep = FLT_EPSILON;
DebugLog(ep); // 1.19209e-07
```

`fenv.h`/`cfenv` gives us fine-grain control over how the CPU deals with floating point numbers. There's no real equivalent to this in C#:

```
#include <fenv.h>

// Clear CPU float exceptions. Different than C++
exceptions.
feclearexcept(FE_ALL_EXCEPT);

// Divide by zero
// Use volatile to prevent the compiler from removing
this
volatile float n = 1.0f;
volatile float d = 0.0f;
volatile float q = n / d;

// Check float exceptions to see if this was a divide by
zero or produced
// an inexact result
int divByZero = fetestexcept(FE_DIVBYZERO);
int inexact = fetestexcept(FE_INEXACT);
DebugLog(divByZero != 0); // true
DebugLog(inexact != 0); // false

// Clear float exceptions
feclearexcept(FE_ALL_EXCEPT);

// Perform a division whose quotient can't be represented
exactly
d = 10.0f;
q = n / d;
```

```
// Check float exceptions
divByZero = fetestexcept(FE_DIVBYZERO);
inexact = fetestexcept(FE_INEXACT);
DebugLog(divByZero != 0); // false
DebugLog(inexact != 0); // true
```

Strings and Arrays

The next category of headers deals with strings and arrays. Let's start with `string.h`/`cstring` which has a lot of operations that are built into the `string` class, managed arrays, and `Buffer` in C#:

```
#include <string.h>

// Compare strings: 0 for equality, -1 for less than, 1
for greater than
DebugLog(strcmp("hello", "hello")); // 0
DebugLog(strcmp("goodbye", "hello")); // -1

// Copy a string
char buf[32];
strcpy(buf, "hello");
DebugLog(buf);

// Concatenate strings
strcat(buf + 5, " world");
DebugLog(buf); // hello world

// Count characters in a string (its length)
// This iterates until NUL is found
DebugLog(strlen(buf)); // 11

// Get a pointer to the first occurrence of a character
in a string
```

```
DebugLog(strchr(buf, 'o')); // o world

// Get a pointer to the first occurrence of a string in a
string
DebugLog(strstr(buf, "ll")); // llo world

// Get a pointer to the next "token" in a string,
separated by a delimiter
// Stores state globally: not thread-safe
char* next = strtok(buf, " ");
DebugLog(next); // hello
next = strtok(nullptr, ""); // null means to continue the
global state
DebugLog(next); // world

// Copy the first three bytes of buf ("hel") to later in
the buffer
memcpy(buf + 3, buf, 3);
DebugLog(buf); // helhelworld

// Set all bytes in buf to 65 and put a NUL at the end
memset(buf, 65, 31);
buf[31] = 0;
DebugLog(buf); // AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

`wchar.h`/`cwchar` is the equivalent for “wide” characters. Support for various character types in C# is provided by the `System.Text` namespace, which has similar functionality to what’s in this header:

```
#include <wchar.h>

wchar_t input[] = L"foo,bar,baz";

// Get the first token, using 'state' to hold the
tokenization state
wchar_t* state;
wchar_t* token = wcstok(input, L",", &state);
DebugLog(token); // foo

// Get the second token
token = wcstok(nullptr, L",", &state);
DebugLog(token); // bar

// Get the third token
token = wcstok(nullptr, L",", &state);
DebugLog(token); // baz
```

`ctype.h`/`cctype` has functions related to just characters. C's lack of a `bool` type means `0` is used instead of `false` and non-`0` is used instead of `true`. C# doesn't use ASCII natively, so this is approximated by `ASCIIEncoding` there:

```
#include <ctype.h>

// Check for alphabetical characters
DebugLog(isalpha('a') != 0); // true
DebugLog(isalpha('9') != 0); // false
```

```
// Check for digit characters
DebugLog(isdigit('a') != 0); // false
DebugLog(isdigit('9') != 0); // true

// Change to uppercase
DebugLog(toupper('a')); // A
```

`wctype.h`/`cwctype` is the equivalent for “wide” characters. A lot of this is built into the `char` type in C#:

```
#include <wctype.h>

// Check for alphabetical characters
DebugLog(iswalpha(L'a') != 0); // true
DebugLog(iswalpha(L'9') != 0); // false

// Check for digit characters
DebugLog(iswdigit(L'a') != 0); // false
DebugLog(iswdigit(L'9') != 0); // true

// Change to uppercase
DebugLog(towupper(L'a')); // A
```

`uchar.h`/`cuchar` has character conversion functions. The “encoding” classes in C#’s `System.Text` namespace provide for these conversions in .NET:

```
// Convert to UTF-16
char input[] = "A";
char16_t output;
mbstate_t state{};
size_t len = mbrtoc16(&output, input, MB_CUR_MAX,
&state);
DebugLog(len); // 1
uint8_t* outputBytes = (uint8_t*)&output;
DebugLog(outputBytes[0], outputBytes[1]); // 65, 0
```

Language Tools

This category of header files includes a range of tools that aren't part of the C or C++ language, but are closely tied to it or would be built into other languages.

First up is `stdarg.h`/`cstdarg`. This header contains the required types and macros to implement variadic function. These are uncommonly used in C++ since variadic templates are available, easier to use, and type-safe. In C#, we'd use the `params` keyword to have the compiler generate a managed array of arguments at the call site. Here's how to use the `va_` macros to implement a variadic function:

```
#include <stdarg.h>

// The "..." indicates a variadic function
void PrintLogs(int count, ...)
{
    // A va_list holds the state
    va_list args;

    // Use the "va_start" macro to start getting args
    va_start(args, count);

    for (int i = 0; i < count; ++i)
    {
        // Use the "va_arg" macro to get the next arg
        const char* log = va_arg(args, const char*);
        DebugLog(log);
    }
```

```
    // Use the "va_end" macro to stop getting args
    va_end(args);
}

// Call the variadic function
PrintLogs(3, "foo", "bar", "baz"); // foo, bar, baz
```

Next there is `assert.h`/`cassert` containing the `assert` macro. If the `NDEBUG` preprocessor symbol is defined, this checks if a condition is false and calls `std::abort` to end the program, possibly with additional debugging steps such as breaking an interactive debugger. If the condition is true, nothing happens. If `NDEBUG` isn't defined, the condition itself is stripped out of the program and not compiled. In C#, we'd use the `[Conditional]` attribute to build an assert or make use of the existing `Debug.Assert`:

```
#include <assert.h>

assert(2 + 2 == 4); // OK
assert(2 + 2 == 5); // Calls std::abort and maybe more
```

Then we have `setjmp.h`/`csetjmp`, used to implement a high-powered version of `goto`. This can jump outside of a function, but by breaking these normal language rules eschews the normal destructor calls that are used to clean up local objects. None of this is available in C#:

```c
#include <setjmp.h>

// Saved execution state
jmp_buf buf;

// Use volatile to prevent the compiler from optimizing this away
volatile int count = 0;

void Goo()
{
    count++;
    DebugLog("Goo calling longjmp with", count);

    // Go to the saved execution state and pass 'count' as the 'status'
    longjmp(buf, count);
}

void Foo()
{
    DebugLog("Foo");

    // Save the execution state
    // When longjmp is called, execution goes here
    // The passed 'status' is "returned" from setjmp
    int status = setjmp(buf);
    DebugLog("Foo got status", status);
```

```
    if (status >= 3)
    {
        return;
    }
    DebugLog("Foo calling Goo");
    Goo();
}
```

This prints the following:

```
Foo
Foo got status, 0
Foo calling Goo
Goo calling longjmp with, 1
Foo got status, 1
Foo calling Goo
Goo calling longjmp with, 2
Foo got status, 2
Foo calling Goo
Goo calling longjmp with, 3
Foo got status, 3
```

Lastly, there's `errno.h`/`cerrno`. This header provides the `errno` macro that holds a global error flag used by several C Standard Library functions. This is generally considered to be a poor way of handling errors as it's not thread-safe and the caller needs to know to check something that isn't part of the function signature. It's never used in C#, so there's really no equivalent. It is widely used in the C Standard Library though, so let's see how it works:

```
#include <errno.h>

// Pass an invalid argument to sqrt (from math.h)
float root = sqrt(-1.0f);

// It returns NaN
DebugLog(root); // NaN

// It signals this error by setting errno to EDOM (out of
domain)
DebugLog(errno); // Maybe 33

// Check that this is what was set
DebugLog(errno == EDOM); // true
```

System Integration

The last category of header files deals with the system on which we run our programs. Let's start with `time.h`/`ctime` which is like a basic version of `DateTime` in C#:

```
#include <time.h>

// Get the time in the return value and in the pointer we
pass
time_t t1{};
time_t t2 = time(&t1);
DebugLog(t1, t2); // Maybe 1612052060, 1612052060

// Get the amount of CPU time the program has used
// Not in relation to any particular time (like the UNIX
epoch)
clock_t c1 = clock();

// Do something expensive we want to benchmark
volatile float f = 123456;
for (int i = 0; i < 1000000; ++i)
{
    f = sqrtf(f);
}

// Check the clock again
clock_t c2 = clock();
```

```
double secs = ((double)(c2) - c1) / CLOCKS_PER_SEC;
DebugLog("Took", secs, "seconds"); // Maybe: Took 0.011
seconds
```

We also have `signal.h`/`csignal` to deal with OS signals. This allows us to deal with signals such as being terminated by the OS and to raise such signals ourselves. This isn't normally done with C# as the .NET environment our program is running in handles such signals:

```
#include <signal.h>

signal(SIGTERM, [](int val){DebugLog("terminated with",
val); });
raise(SIGTERM); // Maybe: terminated with 15
```

Many C Standard Library functions use a global "locale" setting to determine how they work. The `locale.h`/`clocale` header file has functions to change this setting. It's similar to the thread-specific `CultureInfo` in C#:

```
#include <locale.h>

// Set the locale for everything to Japanese
// This is global: not thread-safe
setlocale(LC_ALL, "ja_JP.UTF-8");

// Get the global locale
```

```
lconv* lc = localeconv();
DebugLog(lc->currency_symbol); // ¥
```

And finally, we'll end with the header that enables "Hello, world!" in C: `stdio.h`/`cstdio`. This is like `Console` in C#. There's also file system access, similar to the methods of `File` in C#:

```
#include <stdio.h>

// Output a formatted string to stdout
// The first string is the "format string" with value
placeholders: %s %d
// Subsequent values must match the placeholders' types
// This is a variadic function
printf("%s %d\n", "Hello, world!", 123); // Hello, world!
123

// Read a value from stdin
// The same "format string" is used to accept different
types
int val;
int numValsRead = scanf("%d", &val);
DebugLog(numValsRead); // {1 if the user entered a
number, else 0}
if (numValsRead == 1)
{
    DebugLog(val); // {Number the user typed}
}
```

```
// Open a file, seek to its send, get the position, and close it
FILE* file = fopen("/path/to/myfile.dat", "r");
fseek(file, 0, SEEK_END);
long len = ftell(file);
fclose(file);
DebugLog(len); // {Number of bytes in the file}

// Delete a file
int deleted = remove("/path/to/deleteme.dat");
DebugLog(deleted == 0); // True if the file was deleted

// Rename a file
int renamed = rename("/path/to/oldname.dat", "/path/to/newname.dat");
DebugLog(renamed == 0); // True if the file was renamed
```

Conclusion

The C Standard Library is very old, but still very commonly used. Being so old and based on the much less powerful C, a lot of its design leaves a lot to be desired. The global states used by functions like `rand` and `strtok` and macros like `errno` aren't thread-safe and are difficult to understand how to use correctly. Using special `int` values, even inconsistently, instead of more structured outputs like exceptions and enumerations is similarly difficult to use.

Regardless of any complaints we may have about the C Standard Library's design, we still need to know how to use it. The C++ Standard Library offers alternatives to much of what we've seen here in this chapter, but that's not always the case. Sure, we can swap in `<random>` for `rand`, `<chrono>` for `time`, and `<filesystem>` for `remove`, but `assert` and `stdint.h` remain the most modern standardized ways of achieving those areas of functionality.

From here on we'll be covering the C++ part of the C++ Standard Library. We'll see a lot more modern designs for areas like containers, algorithms, I/O, strings, math, and threading!

39. Language Support Library

Source Location

Let's start with an easy one that was just added in C++20: `<source_location>`. Right away we see how the naming convention of the C++ Standard Library differs from the C Standard Library and it's C++ wrappers. The C Standard Library header file name would likely be abbreviated into something like `srcloc.h`. The C++ wrapper would then be named `csrcloc`. The C++ Standard Library usually prefers to spell out names more verbosely in `snake_case` and without any extension, `.h` or otherwise.

Within the `<source_location>` header file we see the naming convention continue with the `source_location` class. There isn't always a 1:1 mapping, but `snake_case` is almost always used. The `source_location` class is placed in the `std` namespace, so we typically talk about it as `std::source_location`. The `std` namespace is reserved for the C++ Standard Library.

Now to the actual purpose of `std::source_location`. As its name suggests, it provides a facility for expressing a location in the source code. It has copy and move constructors, but no way for us to create one from scratch. Instead, we call its static member function `current` and one is returned:

```
#include <source_location>

void Foo()
{
    std::source_location sl =
std::source_location::current();
```

```
    DebugLog(sl.line()); // 42
    DebugLog(sl.column()); // 61
    DebugLog(sl.file_name()); // example.cpp
    DebugLog(sl.function_name()); // void Foo()
}
```

The `file_name` member function provides a replacement for the `__FILE__` macro. Likewise, `line` replaces `__LINE__`. We also get `column` and `function_name` which aren't present in standardized macro form. In C#, the `StackTrace` and `StackFrame` classes are roughly equivalent to `source_location`.

It's worth noting here at the start that a lot of code will include a `using` statement to remove the need to type `std::` over and over. It's a namespace like any other, so we have all the normal options. For example, a `using namespace std;` at the file level right after the `#include` lines is common:

```
#include <source_location>
using namespace std;

void Foo()
{
    source_location sl = source_location::current();
    DebugLog(sl.line()); // 43
    DebugLog(sl.column()); // 61
    DebugLog(sl.file_name()); // example.cpp
    DebugLog(sl.function_name()); // void Foo()
}
```

To avoid bringing the entire Standard Library into scope, we might `using` just particular classes:

```
#include <source_location>
using std::source_location;

void Foo()
{
    source_location sl = source_location::current();
    DebugLog(sl.line()); // 43
    DebugLog(sl.column()); // 61
    DebugLog(sl.file_name()); // example.cpp
    DebugLog(sl.function_name()); // void Foo()
}
```

Or we might put the `using` just where the Standard Library is being used:

```
#include <source_location>

void Foo()
{
    using namespace std;
    source_location sl = source_location::current();
    DebugLog(sl.line()); // 43
    DebugLog(sl.column()); // 61
    DebugLog(sl.file_name()); // example.cpp
```

```
    DebugLog(sl.function_name()); // void Foo()
}
```

All of these are commonly seen in C++ codebases and provide good options for removing a lot of the `std::` clutter. There is, however, one *bad* option which should be avoided: adding `using` at the top level of header files. Because header files are essentially copied and pasted into other files via `#include`, these `using` statements introduce the Standard Library to name lookup for all the files that `#include` them. When header files `#include` other header files, this impact extends even further:

```
// top.h
#include <source_location>
using namespace std; // Bad idea

// middlea.h
#include "top.h" // Pastes "using namespace std;" here

// middleb.h
#include "top.h" // Pastes "using namespace std;" here

// bottoma.cpp
#include "middlea.h" // Pastes "using namespace std;"
here

// bottomb.cpp
#include "middlea.h" // Pastes "using namespace std;"
here
```

```
// bottomc.cpp
#include "middleb.h" // Pastes "using namespace std;"
here

// bottomd.cpp
#include "middled.h" // Pastes "using namespace std;"
here
```

The effects of `using namespace std;` in `top.h` have spread to the files that `#include` it: `middlea.h` and `middleb.h`. That then spreads to the files that `#include` those: `bottoma.cpp`, `bottomb.cpp`, `bottomc.cpp`, and `bottomd.cpp`. It's best to avoid this to so as to not undo the compartmentalization that namespaces provide and instead let individual files choose when and where they want to breach it:

```
// top.h
#include <source_location>
struct SourceLocationPrinter
{
    static void Print()
    {
        // OK: only applies to this function, not files
that #include
        using namespace std;

        source_location sl = source_location::current();
        DebugLog(sl.line()); // 43
```

```
        DebugLog(sl.column()); // 61
        DebugLog(sl.file_name()); // example.cpp
        DebugLog(sl.function_name()); // void
SourceLocationPrinter::Print()
    }
};

// middlea.h
#include "top.h" // Does not paste "using namespace std;"
here

// middleb.h
#include "top.h" // Does not paste "using namespace std;"
here
```

Initializer List

Next up let's look at `<initializer_list>`. We touched on `std::initializer_list` before, but now we'll take a closer look. An instance of this class template is automatically created and passed to the constructor when we use braced list initialization:

```
struct AssetLoader
{
    AssetLoader(std::initializer_list<const char*> paths)
    {
        for (const char* path : paths)
        {
            DebugLog(path);
        }
    }
};

AssetLoader loader = {
    "/path/to/model",
    "/path/to/texture",
    "/path/to/audioclip"
};
```

We could rewrite this to create the `std::initializer_list<const char*>` manually, but this relies on that same braced list initialization as `std::initializer_list` doesn't have any direct way to create an empty instance:

```
AssetLoader loader(std::initializer_list<const char*>{
    "/path/to/model",
    "/path/to/texture",
    "/path/to/audioclip"
});
```

As we see in the `AssetLoader` constructor, range-based for loops work with `std::initializer_list`. There's also a `size` member function, but there's no index operator so we can't use a typical `for` loop:

```
AssetLoader(std::initializer_list<const char*> paths)
{
    // OK: there's a size member function
    for (size_t i = 0; i < paths.size(); ++i)
    {
        // Compiler error: no operator[int]
        DebugLog(paths[i]);
    }
}
```

The C# equivalent is to take a `params` managed array. The compiler builds that managed array for us at the call site like how a `std::initializer_list` is built for us.

Type Info and Index

We've also seen a little bit of `<typeinfo>` when looking at RTTI. When we use `typeid`, we get back a `std::type_info` which is like a lightweight version of the C# `Type` class:

```
#include <typeinfo>

struct Vector2
{
    float X;
    float Y;
};

struct Vector3
{
    float X;
    float Y;
    float Z;
};

Vector2 v2{ 2, 4 };
Vector3 v3{ 2, 4, 6 };

// All constructors are deleted, but we can still get a
reference
const std::type_info& ti2 = typeid(v2);
const std::type_info& ti3 = typeid(v3);
```

```
// There are only three public members
// They are all implementation-specific
DebugLog(ti2.name()); // Maybe struct Vector2
DebugLog(ti2.hash_code()); // Maybe 3282828341814375180
DebugLog(ti2.before(ti3)); // Maybe true
```

Relatedly, `<typeinfo>` defines the `bad_typeid` class that's thrown as an exception when trying to take the `typeid` of a null pointer to a polymorphic class. In C# we'd get a `NullReferenceException` instead of this when we try to write `nullObj.GetType()`:

```
#include <typeinfo>

struct Vector2
{
    float X;
    float Y;

    // A virtual function makes this class polymorphic
    virtual bool IsNearlyZero(float epsilonSq)
    {
        return abs(X*X + Y*Y) < epsilonSq;
    }
};

void Foo()
{
```

```
    Vector2* pVec = nullptr;
    try
    {
        // Try to take typeid of a null pointer to a
polymorphic class
        DebugLog(typeid(*pVec).name());
    }
    // This particular exception is thrown
    catch (const std::bad_typeid& e)
    {
        DebugLog(e.what()); // Maybe "Attempted a typeid
of nullptr pointer!"
    }
}
```

There's also a `bad_cast` class that's thrown when we try to `dynamic_cast` two unrelated types. This is the equivalent of the C# `InvalidCastException` class:

```
#include <typeinfo>

struct Vector2
{
    float X;
    float Y;

    virtual bool IsNearlyZero(float epsilonSq)
    {
```

```
        return abs(X*X + Y*Y) < epsilonSq;
    }
};

struct Vector3
{
    float X;
    float Y;
    float Z;

    virtual bool IsNearlyZero(float epsilonSq)
    {
        return abs(X*X + Y*Y + Z*Z) < epsilonSq;
    }
};

void Foo()
{
    Vector3 vec3{};
    try
    {
        Vector2& vec2 = dynamic_cast<Vector2&>(vec3);
    }
    catch (const std::bad_cast& e)
    {
        DebugLog(e.what()); // Maybe "Bad dynamic_cast!"!
    }
}
```

The `<typeindex>` header provides the `std::type_index` class, not an integer, which wraps the `std::type_info` we saw above. This class provides some overloaded operators so we can compare them in various ways, not just with the `before` member function:

```
#include <typeindex>

struct Vector2
{
    float X;
    float Y;
};

struct Vector3
{
    float X;
    float Y;
    float Z;
};

Vector2 v2{ 2, 4 };
Vector3 v3{ 2, 4, 6 };

// Pass a std::type_info to the constructor
const std::type_index ti2{ typeid(v2) };
const std::type_index ti3{ typeid(v3) };

// Some member functions from std::type_info carry over
DebugLog(ti2.name()); // Maybe struct Vector2
```

```
DebugLog(ti2.hash_code()); // Maybe 32828283418143751 80

// Overloaded operators are provided for comparison
DebugLog(ti2 == ti3); // false
DebugLog(ti2 < ti3); // Maybe true
DebugLog(ti2 > ti3); // Maybe false
```

The C# `Type` class can't be compared directly, so we'd instead compare something like its fully-qualified name string.

Compare

C++20 introduced the [three-way comparison operator](): `x <=> y`. This allows us to overload one operator stating how our class compares to another class. We need return an object that supports all of the individual comparison operators: `<`, `<=`, `>`, `>=`, `==`, and `!=`. Rather than defining our own class to do that, the Standard Library provides some built-in classes via the `<compare>` header. For example, we can return a `std::strong_ordering` via one of its static data members:

```
#include <compare>

struct Integer
{
    int Value;

    std::strong_ordering operator<=>(const Integer&
other) const
    {
        // Determine the relationship once
        return Value < other.Value ?
            std::strong_ordering::less :
            Value > other.Value ?
                std::strong_ordering::greater :
                std::strong_ordering::equal;
    }
};

Integer one{ 1 };
```

```
Integer two{ 2 };
std::strong_ordering oneVsTwo = one <=> two;

// All the individual comparison operators are supported
DebugLog(oneVsTwo < 0); // true
DebugLog(oneVsTwo <= 0); // true
DebugLog(oneVsTwo > 0); // false
DebugLog(oneVsTwo >= 0); // false
DebugLog(oneVsTwo == 0); // false
DebugLog(oneVsTwo != 0); // true
```

There are similar classes for weaker comparison results: `std::weak_ordering` and `std::partial_ordering`. There are also helper functions that call all of these operators on any of these comparison classes so we can write this instead:

```
DebugLog(std::is_lt(oneVsTwo)); // true
DebugLog(std::is_lteq(oneVsTwo)); // true
DebugLog(std::is_gt(oneVsTwo)); // false
DebugLog(std::is_gteq(oneVsTwo)); // false
DebugLog(std::is_eq(oneVsTwo)); // false
DebugLog(std::is_neq(oneVsTwo)); // true
```

Helper functions are provided to get these ordering class objects, even from primitives:

```
std::strong_ordering so = std::strong_order(1, 2);
std::weak_ordering wo = std::weak_order(1, 2);
```

```
std::partial_ordering po = std::partial_order(1, 2);
std::strong_ordering sof =
std::compare_strong_order_fallback(1, 2);
std::weak_ordering wof =
std::compare_weak_order_fallback(1, 2);
std::partial_ordering pof =
std::compare_partial_order_fallback(1, 2);
```

There's no equivalent to the `<=>` operator in C#, so there's no equivalent to this header.

Concepts

Another C++20 feature with library support is concepts. A whole host of pre-defined concepts are available for our immediate use and for us to extend. Here are a few of them:

```
#include <concepts>

template <typename T1, typename T2>
requires std::same_as<T1, T2>
bool SameAs;

template <typename T>
requires std::integral<T>
bool Integral;

template <typename T>
requires std::default_initializable<T>
bool DefaultInitializable;

SameAs<int, int>; // OK
SameAs<int, float>; // Compiler error

Integral<int>; // OK
Integral<float>; // Compiler error

struct NoDefaultCtor { NoDefaultCtor() = delete; };
```

```
DefaultInitializable<int>; // OK
DefaultInitializable<NoDefaultCtor>; // Compiler error
```

There are many more available for diverse needs: `derived_from`, `destructible`, `equality_comparable`, `copyable`, `invocable`, and so forth. None of these have a C# counterpart as C# generic constraints are not customizable.

Coroutine

The final C++20 feature receiving library support is coroutine. The `<coroutine>` header provides the required `std::coroutine_handle` class we've already seen when implementing our own coroutine "return objects." It also provides `std::suspend_never` and `std::suspend_always` so we don't have to write our own versions as we did before. Here's how our trivial coroutine example would have looked with `std::suspend_never` instead of our custom `NeverSuspend` class:

```
#include <coroutine>

struct ReturnObj
{
    ReturnObj()
    {
        DebugLog("ReturnObj ctor");
    }

    ~ReturnObj()
    {
        DebugLog("ReturnObj dtor");
    }

    struct promise_type
    {
        promise_type()
        {
```

```
        DebugLog("promise_type ctor");
    }

    ~promise_type()
    {
        DebugLog("promise_type dtor");
    }

    ReturnObj get_return_object()
    {
        DebugLog("promise_type::get_return_object");
        return ReturnObj{};
    }

    std::suspend_never initial_suspend()
    {
        DebugLog("promise_type::initial_suspend");
        return std::suspend_never{};
    }

    void return_void()
    {
        DebugLog("promise_type::return_void");
    }

    std::suspend_never final_suspend()
    {
        DebugLog("promise_type::final_suspend");
```

```
            return std::suspend_never{};
        }

        void unhandled_exception()
        {
            DebugLog("promise_type unhandled_exception");
        }
    };
};

ReturnObj SimpleCoroutine()
{
    DebugLog("Start of coroutine");
    co_return;
    DebugLog("End of coroutine");
}

void Foo()
{
    DebugLog("Calling coroutine");
    ReturnObj ret = SimpleCoroutine();
    DebugLog("Done");
}
```

Here's what this prints:

```
Calling coroutine
promise_type ctor
```

```
promise_type::get_return_object
ReturnObj ctor
promise_type::initial_suspend
Start of coroutine
promise_type::return_void
promise_type::final_suspend
promise_type dtor
Done
ReturnObj dtor
```

There's also a trio of no-op coroutine features: `std::noop_coroutine`, `std::noop_coroutine_promise`, and `std::noop_coroutine_handle`. These implement the coroutine equivalent of a `void noop() {}` function. `noop_coroutine` is the coroutine and it returns a `noop_coroutine_handle` whose "promise" is a `noop_coroutine_promise`.

C# doesn't provide this level of customization for its iterator functions, but we can implement `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, and `IEnumerator<T>` to take some control over iteration. Those interfaces and their methods provide the closest analog to this header file.

Version

As we saw when looking at the preprocessor, a `<version>` header exists with a ton of macros we can use to check if various features are available in the language and Standard Library. For example, we can check for some of the Standard Library features we've seen in this chapter:

```
#include <version>

// These print "true" or "false" depending on whether the
Standard Library has
// these features available
DebugLog("Standard Library concepts?", __cplusplus >=
__cpp_lib_concepts);
DebugLog("source_location?", __cplusplus >=
__cpp_lib_source_location);
```

C# has a handful of standardized preprocessor symbols, including `DEBUG` and `TRACE`, but its suite is nowhere near as extensive as in C++. Each .NET implementation, such as Unity and .NET Core, may define its own additional symbols, such as `UNITY_2020_2_OR_NEWER` and these version numbers are often correlated to available language and library features.

Type Traits

Finally for this chapter we have `<type_traits>` which is used for compile-time programming. This header predates concepts in C++20, so a lot of it overlaps in a non-concept form. For example, we have various `constexpr` variable templates that check whether types fulfill certain criteria. These are available as `static` member variables of class templates and as namespace-scope variable templates:

```
#include <type_traits>

// Use a static value data member of a class template
static_assert(std::is_integral<int>::value); // OK
static_assert(std::is_integral<float>::value); //
Compiler error

// Use a variable template
static_assert(std::is_integral_v<int>); // OK
static_assert(std::is_integral_v<float>); // Compiler
error
```

There are tons of these available and they can check for nearly any feature of a type. Here are some more advanced ones:

```
#include <type_traits>

struct Vector2
{
```

```
    float X;
    float Y;
};

struct Player
{
    int Score;

    Player(const Player& other)
    {
        Score = other.Score;
    }
};

static_assert(std::is_bounded_array_v<int[3]>); // OK
static_assert(std::is_bounded_array_v<int[]>); //
Compiler error

static_assert(std::is_trivially_copyable_v<Vector2>); //
OK
static_assert(std::is_trivially_copyable_v<Player>); //
Compiler error
```

Besides type checks, there are various utilities for querying types:

```
#include <type_traits>

DebugLog(std::rank_v<int[10]>); // 1
```

```
DebugLog(std::rank_v<int[10][20]>); // 2
DebugLog(std::extent_v<int[10][20], 0>); // 10
DebugLog(std::extent_v<int[10][20], 1>); // 20
DebugLog(std::alignment_of_v<float>); // Maybe 4
DebugLog(std::alignment_of_v<double>); // Maybe 8
```

We can also get modified versions of types:

```
#include <type_traits>

// We know T is a pointer (e.g. int*)
// We don't have a name for what it points to (e.g. int)
// Use std::remove_pointer_t to get it
template <typename T>
auto Dereference(T ptr) -> std::remove_pointer_t<T>
{
    return *ptr;
}

int x = 123;
int* p = &x;
int result = Dereference(p);
DebugLog(result); // 123
```

One particularly useful function is `std::underlying_type` which can be used to implement safe “cast” functions to and from enumerations:

```cpp
#include <type_traits>

// "Cast" from an integer to an enum
template <typename TEnum, typename TInt>
TEnum FromInteger(TInt i)
{
    // Make sure the template parameters are an enum and an integer
    static_assert(std::is_enum_v<TEnum>);
    static_assert(std::is_integral_v<TInt>);

    // Use is_same_v from type_traits to ensure that TInt is the underlying type
    // of TEnum

static_assert(std::is_same_v<std::underlying_type_t<TEnum>, TInt>);

    // Perform the cast
    return static_cast<TEnum>(i);
}

// "Cast" from an enum to an integer
template <typename TEnum>
auto ToInteger(TEnum e) -> std::underlying_type_t<TEnum>
{
    // Make sure the template parameter is an enum
    static_assert(std::is_enum_v<TEnum>);
```

```
    // Perform the cast
    return static_cast<std::underlying_type_t<TEnum>>(e);
}

enum class Color : uint64_t
{
    Red,
    Green,
    Blue
};

Color c = Color::Green;
DebugLog(c); // Green

// Cast from enum to integer
auto i = ToInteger(c);
DebugLog(i); // 1

// Cast from integer to enum
Color c2 = FromInteger<Color>(i);
DebugLog(c2); // Green
```

These “cast” functions imply no runtime overhead as all the checks occur at compile time. They do, however, add safety since we’ll get a compiler diagnostic if we accidentally try to use the wrong size of type:

```
// Compiler error: short is not the underlying type
FromInteger<Color>(uint16_t{ 1 });

// Compiler warning: target integer type is too small
// The "treat warnings as errors" setting can be used to
turn this into an error
uint16_t i = ToInteger(c);
```

Some of this functionality exists in C# via the `Type` class and its related reflection classes: `FieldInfo`, `PropertyInfo`, etc. In contrast to C++, these all execute at runtime where their C++ counterparts execute at compile time.

Conclusion

Some parts of C++ rely on the Standard Library. We need to use `std::initializer_list` to handle braced list initialization and we need to use `std::coroutine_handle` to implement coroutine return objects. This is similar to C# that enshrines parts of the .NET API into the language: `Type`, `System.Single`, etc.

In this chapter we've seen a lot of those quasi-language features as well as some general language support functionality like `source_location` and a lot of pre-defined concepts. These are foundational elements of the language and library, but also give a taste of what's to come in terms of the Standard Library's design.

40. Utilities Library

Exception

Let's start by looking at how the Standard Library codifies error-handling. There are a lot of kinds of errors from a lot of sources that can be dealt with in a lot of ways, so it's no surprise that the Standard Library provides a lot of different approaches to error-handling.

To begin, let's look at the `<exception>` header. As in C#, exceptions are the primary error-handling approach in C++. As C# has `System.Exception` as the base class of all exceptions, C++ has `std::exception` as the base class of all Standard Library exceptions. We're free to throw anything, not just `std::exception`, but the Standard Library only throws this type and and many C++ codebases do the same.

```
#include <exception>

// Derive our own exception type
struct MyException : public std::exception
{
    const char* msg;

    MyException(const char* msg)
        : msg(msg)
    {
    }
```

```
    virtual const char* what() const noexcept override
    {
        return msg;
    }
};

try
{
    throw MyException{ "boom" };
}
catch (const std::exception& ex)
{
    DebugLog(ex.what()); // boom
}
```

This shows the standard usage pattern of C++ exceptions. When throwing, we throw an object as opposed to a pointer to an object allocated with the `new` operator. When catching, we catch by `const` lvalue reference. This avoids making a copy of the exception object and avoids accidentally changing the exception in the `catch` block.

All `std::exception` objects have a `virtual` `what()` function returning an error message just as `System.Exception` has a `Message` property in C#. Both C++ and C# have many types derived their base exception classes to provide additional detail about the error. This is done via the type system as well as the possibility of adding additional members to the derived types. We'll see some of those later in this chapter.

C++ provides a way to capture these `std::exception` objects so we can deal with them later. For example, we might want to catch

exceptions on one thread and re-throw them on another to provide thread-safety.

```
#include <exception>

// A class that acts like a pointer to a captured
exception
std::exception_ptr capturedEx;

try
{
    // Do something that throws
    throw MyException{ "boom" };
}
// Catch anything
catch (...)
{
    // Capture the current exception
    capturedEx = std::current_exception();
}

// Later...
try
{
    // Check if an exception was captured
    if (capturedEx)
    {
        // If so, re-throw it
```

```
        std::rethrow_exception(capturedEx);
    }
}
catch (const std::exception& ex)
{
    DebugLog(ex.what()); // boom
}
```

There's also a way to nest exceptions within each other:

```
#include <exception>

// Recursively print an exception and all its nested
exceptions
void PrintNestedExceptions(const std::exception& ex)
{
    DebugLog(ex.what());

    try
    {
        // If ex is a std::nested_exception, re-throw its
nested std::exception
        // Otherwise do nothing
        std::rethrow_if_nested(ex);
    }
    catch (const std::exception& nestedEx)
    {
        // Recurse to print the nested exception (and its
```

```
nested exceptions)
        PrintNestedExceptions(nestedEx);
    }
}

// Function that throws an exception
FILE* OpenFile(const char* path)
{
    FILE* handle = fopen(path, "r");
    if (!handle)
    {
        throw MyException{ "Error opening file" };
    }
    return handle;
}

// Function that calls a function that throws an
exception
// It throws an exception with the caught exception
nested
void PrintFirstByte(const char* path)
{
    try
    {
        // Call a function that throws an exception
        FILE* f = OpenFile(path);

        DebugLog("First byte:", fgetc(f));
```

```
        fclose(f);
    }
    // Catch OpenFile exceptions
    catch (...)
    {
        // Throw an exception with the caught exception
nested in it
        std::throw_with_nested(MyException{ "Failed to
read file" });
    }
}

try
{
    // Call a function that throws a
std::nested_exception
    PrintFirstByte("/path/to/missing/file");
}
// Catch all std::exception objects
// Includes the derived std::nested_exception type
catch (const std::exception& ex)
{
    PrintNestedExceptions(ex);
}
```

We have ways to customize what happens when `std::terminate` or `std::unexpected` are called. The language says that `std::terminate` is called for a variety of reasons including a `noexcept` function throwing an exception or an exception that's never caught. The

`std::unexpected` function was removed in C++17, but it was previously called when a dynamic exception specification (`throw(MyException)`) was violated. Dynamic exception specifications were also removed in C++17.

```
#include <exception>

// Set a lambda to be called when std::terminate is
called
std::set_terminate([]() { DebugLog("std::terminate
called"); });

// Throw an exception and never catch it
// This causes std::terminate to be called
// The lambda is then called
throw MyException{ "boom" };
```

And finally, we can use `std::uncaught_exceptions` to check how many exceptions have been thrown that haven't yet been caught by a `catch` block. A singular version, `std::uncaught_exception`, was available until C++20 when it was removed. Multiple exceptions can be uncaught when destructors throw exceptions themselves and the plural `std::uncaught_exceptions` allows us to check for that:

```
#include <exception>

struct Second
{
    ~Second()
```

```
    {
        DebugLog("Second", std::uncaught_exceptions());
    }
};

struct First
{
    ~First()
    {
        DebugLog("First before",
std::uncaught_exceptions());

        try
        {
            Second sec;
            throw std::runtime_error{ "boom" };
        } // Note: sec destructor called
        catch (const std::exception& e)
        {
            DebugLog("First caught", e.what());
        }

        DebugLog("First after",
std::uncaught_exceptions());
    }
};

void Foo()
```

```
{
    try
    {
        First fir;
        throw std::runtime_error{ "boom" };
    } // Note: fir destructor called
    catch (const std::exception& e)
    {
        DebugLog("Foo", e.what()); // boom
    }

    First fir2;
} // Note: fir2 destructor called
```

This prints the following:

```
First before 1
Second 2
First caught boom
First after 1
Foo boom
First before 0
Second 1
First caught boom
First after 0
```

Standard Exceptions

Now let's look at some of the classes that derive from `std::exception` to describe particular categories of errors. These are available in `<stdexcept>`:

```
#include <stdexcept>

int GetLastElement(int* array, int length)
{
    if (array == nullptr || length <= 0)
    {
        // C# approximation: ArgumentException
        throw std::invalid_argument{ "Invalid array" };
    }

    return array[length - 1];
}

float Sqrt(float val)
{
    if (val < 0)
    {
        // C# approximation: ArgumentNullException,
DivideByZeroException, etc.
        throw std::domain_error{ "Value must be non-
negative" };
    }
```

```
    return std::sqrt(val);
}

template <typename T, int N>
void WriteToBuffer(const T& obj, char buf[N])
{
    if (sizeof(T) > N)
    {
        // C# approximation: ArgumentException
        throw std::length_error{ "Object is too big for
the buffer" };
    }

    std::memcpy(buf, &obj, sizeof(T));
}

void CheckedIncrement(uint32_t& x)
{
    if (x == 0xffffffff)
    {
        // C# approximation: ArgumentException
        throw std::out_of_range{ "Overflow" };
    }
    x++;
}

int BinarySearch(int* array, int length)
```

```
{
    #if NDEBUG
        for (int i = 1; i < length; ++i)
        {
            if (array[i - 1] > array[i])
            {
                // C# approximation: ArgumentException
                // Note: base class of all of the above
                throw std::logic_error{ "Array isn't
sorted" };
            }
        }
    #endif

    // ...implementation...
}
```

The Standard Library itself throws these exception types. We're also free to throw them in our own code and it's common to do so.

System Error

Next up, let's look at the `<system_error>` header. As we saw in the C Standard Library, there are a lot of "error codes" exposed to us via mechanisms like return values and the global `errno` macro. These error codes are platform-specific. The C++ Standard Library includes a platform-independent alternative in a pair of types: `std::error_condition` and `std::error_category`.

```
#include <system_error>

// Get the "generic" error category
const std::error_category& category =
std::generic_category();
DebugLog(category.name()); // Maybe "generic"

// Build an error_condition representing the "no space on
device" code
std::error_condition condition =
category.default_error_condition(ENOSPC);
DebugLog(condition.value() == ENOSPC); // true
DebugLog(condition.message()); // Maybe "no space on
device"

// There are other categories
const std::error_category& sysCat =
std::system_category();
DebugLog(sysCat.name()); // Maybe "system"
```

We use these classes to convert platform-specific error codes to platform-independent error codes and then take action on them. We get the added bonus of stronger typing since these classes aren't simply an `int` and are therefore less likely to be misused.

The reverse is also supported. We have the `value` member function above to get platform-specific error codes back out of platform-independent `std::error_condition` objects. We also have `std:errc` which is an `enum class` that allows us to avoid macros like `ENOSPC` and strongly-type error codes as opposed to simple `int` values. These are often attached to a `std::system_exception` type derived from `std::exception`:

```
#include <system_error>

try
{
    // Handle a platform-specific error: ENOSPC
    throw std::system_error{
        ENOSPC, std::generic_category(), "Disk is full"
};
}
catch (const std::system_error& e)
{
    // Platform-specific error (ENOSPC) converted to a
std::errc enumerator
    DebugLog(e.code() == std::errc::no_space_on_device);
// true

    DebugLog(e.what()); // Maybe "Disk is full: no space
```

```
on device”
}
```

Utility

Moving on from error-handling, let's look at some truly generic utility functions provied by the `<utility>` header:

```cpp
#include <utility>

// Swap two values
int x = 2;
int y = 4;
std::swap(x, y);
DebugLog(x, y); // 4, 2

// Set a value and return the old value
int old = std::exchange(x, 6);
DebugLog(x, old); // 6, 4

// Get a const version of anything
const int& c = std::as_const(x);
DebugLog(c); // 6

// Compare integers without conversion
DebugLog(-1 > 1U); // true!
DebugLog(std::cmp_greater(-1, 1U)); // false

// Check if an integer fits in an integer type
DebugLog(std::in_range<uint8_t>(200)); // true
DebugLog(std::in_range<uint8_t>(500)); // false
```

```
// Cast to an rvalue reference
int&& rvr = std::move(x);
DebugLog(x, rvr); // 6, 6

// Forward a value as an lvalue or rvalue reference
int f1 = std::forward<int&>(x);
int f2 = std::forward<int&&>(x);
```

There are also a couple of utility classes available. First, we have `std::integer_sequence` to deal with parameter packs of integers:

```
#include <utility>

// Variadic function template taking a
std::integer_sequence
template<typename T, T... vals>
void PrintInts(std::integer_sequence<T, vals...> is)
{
    // Provides the number of integers
    DebugLog(is.size());

    // Use the parameter pack to get the values
    DebugLog(vals...);
}

// Prints "3" then "123, 456, 789"
```

```
PrintInts(std::integer_sequence<int32_t, 123, 456, 789>
{});
```

Next we have `std::pair` which is a struct holding two values. This is similar to `KeyValuePair` in C#:

```
#include <utility>

// Make a struct with an int and a float as non-static
data members
std::pair<int, float> p{ 123, 3.14f };

// Get them in two ways
DebugLog(p.first, p.second); // 123, 3.14
DebugLog(std::get<0>(p), std::get<1>(p)); // 123, 3.14

// We can also use std::make_pair to use type deduction
to avoid
// specifying the types ourselves
p = std::make_pair(123, 3.14f);
DebugLog(p.first, p.second); // 123, 3.14

// make_pair is less necessary in C++17 with template
argument deduction
std::pair p2{ 456, 2.2f };
DebugLog(p2.first, p2.second); // 456, 2.2

// std::swap works with std::pair
```

```
std::swap(p, p2);
DebugLog(p.first, p.second); // 456, 2.2
DebugLog(p2.first, p2.second); // 123, 3.14
```

Tuple

`std::pair` has largely been eclipsed by the more generic `std::tuple` in `<tuple>`. It can hold any number of data members, not just two. This is like the `ValueTuple` family of classes in C#: `ValueTuple<T>`, `ValueTuple<T1, T2>`, `ValueTuple<T1, T2, T3>`, etc. There's only one class template in C++ since variadic templates are supported, so truly any number of data members may be added to a `std::tuple`:

```
#include <tuple>

// Make a struct with an int and a float as non-static
data members
std::tuple<int, float> t{ 123, 3.14f };

// Get them, but only with std::get since there are no
names
DebugLog(std::get<0>(t), std::get<1>(t)); // 123, 3.14

// We can also use std::make_tuple to use type deduction
to avoid
// specifying the types ourselves
t = std::make_tuple(123, 3.14f);
DebugLog(std::get<0>(t), std::get<1>(t)); // 123, 3.14

// make_tuple is less necessary in C++17 with template
argument deduction
std::tuple t2{ 456, 2.2f };
DebugLog(std::get<0>(t2), std::get<1>(t2)); // 456, 2.2
```

```
// std::swap works with std::tuple
std::swap(t, t2);
DebugLog(std::get<0>(t), std::get<1>(t)); // 456, 2.2
DebugLog(std::get<0>(t2), std::get<1>(t2)); // 123, 3.14
```

`std::tuple` has some extended functionality beyond what's provided for `std::pair`:

```
#include <tuple>

std::tuple t{ 123, 3.14f, "hello" };

// Get the number of elements in the tuple at compile
time
constexpr std::size_t size =
std::tuple_size_v<decltype(t)>;
DebugLog(size); // 3

// Get the type of an element of the tuple
std::tuple_element_t<1, decltype(t)> second = std::get<1>
(t);
DebugLog(second); // 3.14

// Create a tuple of lvalue references to variables
int i = 456;
float f = 2.2f;
std::tuple tied = std::tie(i, f);
```

```
i = 100;
f = 200;
DebugLog(std::get<0>(tied), std::get<1>(tied)); // 100,
200

// Convert from std::pair to std::tuple
std::pair p{ 2, 4 };
std::tuple t2{ 0, 0 };
t2 = p;
DebugLog(std::get<0>(t2), std::get<1>(t2)); // 2, 4

// Concatenate tuples
std::tuple<
    // Types from t
    int, float, const char*,
    // Types from tied
    int, float,
    // Types from t2
    int, int
> cat = std::tuple_cat(t, tied, t2);
DebugLog(
    std::get<0>(cat),
    std::get<1>(cat),
    std::get<2>(cat),
    std::get<3>(cat),
    std::get<4>(cat),
    std::get<5>(cat),
    std::get<6>(cat)); // 123, 3.14, hello, 100, 200, 2,
```

```
4

struct IntVector
{
    int X;
    int Y;
};

// Instantiate a class by passing the data members of a
tuple to a
// constructor of that class
IntVector iv = std::make_from_tuple<IntVector>(t2);
DebugLog(iv.X, iv.Y); // 2, 4

// Make a function call, passing the data members of a
tuple as arguments
DebugLog(std::apply([](int a, int b) { return a + b; },
t2)); // 6
```

Unlike C#, there's no way to name the data members of a C++ `std::tuple`. We simply refer to them by index similar to using the default `Item1`, `Item2`, etc. names in C#.

Variant

The `<variant>` header provides `std::variant`, which is essentially a generic tagged union. It holds one of many types and is as big as the largest of them. This type is very useful to pass, return, or hold one of many types. There's no similar type in C#, but we can create our own family of them.

```
#include <variant>

// Make a variant that holds either an int32_t or a
double
// Start off holding an int32_t
std::variant<int32_t, double> v{ 123 };
DebugLog(std::get<int32_t>(v)); // 123
DebugLog(v.index()); // 0

// Switch to holding a double
v = 3.14;
DebugLog(std::get<double>(v)); // 3.14
DebugLog(v.index()); // 1

// Trying to get a type that's not current throws an
exception
DebugLog(std::get<int>(v)); // throws
std::bad_variant_access

// Check the type before getting it
if (std::holds_alternative<int32_t>(v))
```

```
{
    DebugLog("int32_t", std::get<int32_t>(v)); // not
printed
}
else
{
    DebugLog("double", std::get<double>(v)); // double
3.14
}

// Get an int32_t pointer if that's the current type
// If it's not, get nullptr
if (int32_t* pVal = std::get_if<int32_t>(&v))
{
    DebugLog(*pVal); // not printed
}
else
{
    DebugLog("not an int"); // printed
}

// These helpers are common boilerplate to use lambdas
with std::visit
// They're usually stashed away in some "utilities"
header file
template<class... TFuncs> struct overloaded : TFuncs...
{
    using TFuncs::operator()...;
```

```
};
template<class... TFuncs> overloaded(TFuncs...) ->
overloaded<TFuncs...>;

// Call the appropriate lambda for the variant's current
type
std::visit(overloaded {
    [](double val) { DebugLog("double", val); }, //
double 3.14
    [](int32_t val) { DebugLog("int32_t", val); } // not
printed
    }, v);

// A class without a default constructor
struct IntWrapper
{
    int Val;

    IntWrapper(int val)
        : Val(val)
    {
    }
};

// Compiler error: first type needs to be default
constructible
std::variant<IntWrapper, float> v2;
```

```
// No compiler error: std::monostate is default
constructible
// It's just a placeholder to work around this issue
std::variant<std::monostate, IntWrapper, float> v2;

// We can get a monostate, but it has no members so
there's no reason to
std::monostate m = std::get<std::monostate>(v2);
```

Optional

Similar to `std::variant` holding one of many types, `std::optional` holds either a value or the absence of a value. It's similar to `Nullable<T>`/`T?` in C# as well as Optional.

```
#include <optional>

// Create an optional with a value
std::optional<float> f{ 3.14f };

// Dereference it like a pointer to get its value
DebugLog(*f); // 3.14

// By default it has no value
std::optional<float> f2;

// Dereferencing without a value is undefined behavior
DebugLog(*f2); // Could be anything!

// Manually check for a value
if (f2.has_value())
{
    DebugLog(*f2); // not printed
}
else
{
    DebugLog("no value"); // gets printed
```

```
}

// Can also check by converting to bool
if (f2)
{
    DebugLog(*f2); // not printed
}
else
{
    DebugLog("no value"); // gets printed
}

// The value member function throws an exception if
there's no value
DebugLog(f2.value()); // Throws std::bad_optional_access

// Get the value or a default
DebugLog(f2.value_or(0)); // 0

// Assign a value
f2 = 2.2f;
DebugLog(*f2); // 2.2

// Clear a value
f2.reset();
DebugLog(f2.has_value()); // false

// The nullopt constant indicates "no option"
```

```
f = std::nullopt;
DebugLog(f.has_value()); // false
```

Any

Similar to C#'s base `System.Object`/`object` type, C++ has `std::any` in the `<any>` header. This is a container for any type of object or, similar to `null`, no object at all.

```
#include <any>

// Create an empty std::any
std::any a;

// Check whether it has a value or is empty
if (a.has_value())
{
    DebugLog("has value"); // not printed
}
else
{
    DebugLog("empty"); // gets printed
}

// Set its value
a = 3.14f;

// Check the type
DebugLog(a.type() == typeid(float)); // true

// Get the value
```

```
// Note: not a real cast. Just a function with "cast" in
the name.
DebugLog(std::any_cast<float>(a)); // 3.14

// Getting the wrong type throws an exception
try
{
    DebugLog(std::any_cast<int32_t>(a));
}
catch (const std::bad_any_cast& ex)
{
    DebugLog(ex.what()); // Maybe "Bad any_cast"
}

// Destroy the value and go back to being empty
a.reset();
DebugLog(a.has_value()); // false

// Another way to create a std::any
a = std::make_any<int32_t>(123);
DebugLog(std::any_cast<int32_t>(a)); // 123
```

Bit Set

C# has `BitArray` to represent an array of bits. The C++ equivalent is `std::bitset` which is a class templated on the number of bits it holds:

```
#include <bitset>

// Holds three bits that are all zero
std::bitset<3> zeroes;

// Indexing gives us bool values
DebugLog(zeroes[0], zeroes[1], zeroes[2]); // false,
false, false

// Get a bit, but throw an exception if out of bounds
DebugLog(zeroes.test(1)); // false
//DebugLog(zeroes.test(3)); // throws std::out_of_range

// Manually bounds-check against the number of bits
if (3 < zeroes.size())
{
    DebugLog(zeroes[3]); // not printed
}
else
{
    DebugLog("out of bounds"); // gets printed
}
```

```
// Convert the bits of an unsigned long to a bitset
std::bitset<3> bits{ 0b101ul };
DebugLog(bits[0], bits[1], bits[2]); // true, false, true

// Compare bitsets
DebugLog(zeroes == bits); // false

// Check all the bits against 1
DebugLog(bits.all()); // false
DebugLog(bits.any()); // true
DebugLog(bits.none()); // false
DebugLog(bits.count()); // 2

// Set a bit
bits.set(0, false);
DebugLog(bits[0], bits[1], bits[2]); // false, false,
true

// Set all bits to true or false
bits.set();
DebugLog(bits[0], bits[1], bits[2]); // true, true, true
bits.reset();
DebugLog(bits[0], bits[1], bits[2]); // false, false,
false

// Perform bit operations on the set
bits |= 0b010;
```

```
DebugLog(bits[0], bits[1], bits[2]); // false, true,
false
bits >>= 1;
DebugLog(bits[0], bits[1], bits[2]); // true, false,
false

// Get bits as an integer
unsigned long ul = bits.to_ulong();
DebugLog(ul); // 1

// Bits are represented in a compact manner
std::bitset<1024> kb;
DebugLog(sizeof(kb)); // 128
```

Functional

Finally for this chapter, `<functional>` contains function-related utilities. Some of these are class templates that have an `operator()` so they can be called like functions. These were more useful before lambdas were introduced to the language, but still commonly seen as a named shorthand alternative to them:

```
#include <functional>

// An object to perform +
std::plus<int32_t> add;
DebugLog(add(2, 3)); // 5

// An object to perform ==
std::equal_to<int32_t> equal;
DebugLog(equal(2, 2)); // true

// An object to perform ||
std::logical_and<int32_t> la;
DebugLog(la(1, 0)); // false

// An object to perform |
std::bit_and<int32_t> ba;
DebugLog(ba(0b110, 0b011)); // 2 (0b010)

// An object to perform the negation of another object
auto ne = std::not_fn(equal);
DebugLog(ne(2, 3)); // true
```

```
// Some class with a member function
struct Adder
{
    int32_t AddOne(int32_t val)
    {
        return val + 1;
    }
};

// An object to call a member function
auto addOne = std::mem_fn(&Adder::AddOne);
Adder adder;
DebugLog(addOne(adder, 2)); // 3
```

A `std::function` class template is provided to encapsulate any kind of callable object including lambdas, free functions, and function objects. This is similar to delegates like `Action` or `Func` in C#, except that it represents only one function:

```
#include <functional>

// Create a std::function that calls a lambda
std::function<int32_t(int32_t, int32_t)> add{
    [](int32_t a, int32_t b) {return a + b; } };
DebugLog(add(2, 3)); // 5

// Create a std::function that calls a free function
```

```
int32_t Add(int32_t a, int32_t b)
{
    return a + b;
}
std::function<int32_t(int32_t, int32_t)> add2{Add};
DebugLog(add2(2, 3)); // 5

// Create a std::function that calls operator() on a
class object
struct Adder
{
    int32_t operator()(int32_t a, int32_t b)
    {
        return a + b;
    }
};
std::function<int32_t(int32_t, int32_t)> add3{ Adder{} };
DebugLog(add3(2, 3)); // 5
```

Similarly, `std::bind` also creates a callable object by “binding” one or more values and placeholder parameters to it:

```
#include <functional>

// Create an object that calls a lambda
auto add = std::bind(
    // Lambda to call
    [](int32_t a, int32_t b) { return a + b; },
```

```
    // Placeholders for parameters
    std::placeholders::_1,
    std::placeholders::_2);
DebugLog(add(2, 3)); // 5

// Create an object that calls a free function
int32_t Add(int32_t a, int32_t b)
{
    return a + b;
}
auto add2 = std::bind(
    // Free function to call
    Add,
    // Placeholders for parameters
    std::placeholders::_1,
    std::placeholders::_2);
DebugLog(add2(2, 3)); // 5

// Create an object that calls a member function
struct Adder
{
    int32_t Add(int32_t a, int32_t b)
    {
        return a + b;
    }
};
Adder adder;
auto add3 = std::bind(
```

```
    // Member function to call
    &Adder::Add,
    // Object to call it on
    &adder,
    // Placeholders for parameters
    std::placeholders::_1,
    std::placeholders::_2);
DebugLog(add3(2, 3)); // 5
```

In C++20, `std::bind_front` is available as a simpler alternative. It doesn't support more complex options like out-of-order placeholders:

```
// Create an object that calls a lambda
auto add = std::bind_front(
    [](int32_t a, int32_t b) { return a + b; });
DebugLog(add(2, 3)); // 5

// Create an object that calls a free function
auto add2 = std::bind_front(Add);
DebugLog(add2(2, 3)); // 5

// Create an object that calls a member function
Adder adder;
auto add3 = std::bind_front(&Adder::Add, &adder);
DebugLog(add3(2, 3)); // 5
```

And finally, `std::reference_wrapper` stores a reference in a normal class object:

```cpp
#include <functional>

// An integer and a reference to it
int x = 123;
int& r = x;

// Use std::ref to get a reference to it
std::reference_wrapper<int> w = std::ref(r);

// Can copy the wrapper without changing the reference
std::reference_wrapper<int> w2 = w;

// Unwrap the references
DebugLog(w.get(), w2.get()); // 123, 123

// Modifying one modifies the other
int& r2 = w2.get();
r2 = 456;
DebugLog(w.get(), w2.get()); // 456, 456

// std::cref gets a constant reference
std::reference_wrapper<const int> cw = std::cref(x);
cw.get() = 1000; // Compiler error: cw.get() is "const
int&"
```

Conclusion

The C++ Standard Library provides a lot of utility functions and types. Most of them are templates, which is why the Standard Library is often called the Standard *Template* Library or STL.

We have a wide variety of error-handling utilities including platform-independent error codes and an inheritance tree of standardized exceptions akin to `System.Exception` in C#.

There are many general-purpose types available, too. `std::optional` gives us the ability to indicate that a value may or may not be present which is especially useful as a return value of functions that may fail to have a usable result. `std::variant` implements any tagged union for us and is a useful alternative to traditional inheritance trees. `std::any` takes the place of `System.Object` in C# for times where a value really could be any kind of type.

We've even got a lot of function-related tools like `std::function` to abstract the specific kind of function, as C# delegates do. We have many "callable" struct types as associated tools such as `std::bind` and named types like `std::plus`.

As we continue through the Standard Library, we'll keep seeing utilities like `std::exception` crop up again and again.

41. System Integration Library' href

Limits

C# primitive type structs have `const` fields indicating their range: `int.MinValue` and `int.MaxValue`. Likewise, the C++ Standard Library's `<limits>` header provides the `std::numeric_limits` class template. At its core, this provides a type-safe version of the macros in the C Standard Library's `<limits.h>`/`<climits>` and `<stdint.h>`/`<cstdint>`:

```
#include <limits>

DebugLog(std::numeric_limits<int32_t>::min()); //
-2147483648
DebugLog(std::numeric_limits<int32_t>::max()); //
2147483647
```

The `min` and `max` member functions are `constexpr`, so they can be used in compile-time programming just like the equivalent C# `const` fields.

There are a ton more functions and constants available in `numeric_limits`. Here's a selection of them:

```
#include <limits>

// Difference between 1.0 and the next representable
floating-point value
```

```
DebugLog(std::numeric_limits<float>::epsilon()); //
1.19209e-07

// Largest error in rounding a floating-point value
DebugLog(std::numeric_limits<float>::round_error()); //
0.5

// Floating-point constants
DebugLog(std::numeric_limits<float>::infinity()); // inf
DebugLog(std::numeric_limits<float>::quiet_NaN()); // nan
DebugLog(std::numeric_limits<float>::signaling_NaN()); //
nan

// Type info useful when writing templates
DebugLog(std::numeric_limits<float>::is_integer); //
false
DebugLog(std::numeric_limits<float>::is_exact); // false
DebugLog(std::numeric_limits<float>::is_modulo); // false
DebugLog(std::numeric_limits<float>::digits10); // 6
```

Numbers

The `<numbers>` header was introduced in C++20 to provide mathematical constants in the `std::numbers` namespace. C# has a few of these as `const` fields of `Math`, but the selection is limited and only `double` values are provided. C++ provides a more robust set as variable templates for each numeric type:

```
#include <numbers>

// Base 2 log of e
DebugLog(std::numbers::log2e_v<float>); // 1.4427

// Base 10 log of e
DebugLog(std::numbers::log10e_v<float>); // 0.434294

// Pi
DebugLog(std::numbers::pi_v<float>); // 3.14159

// 1 divided by pi
DebugLog(std::numbers::inv_pi_v<float>); // 0.31831

// 1 divided by the square root of pi
DebugLog(std::numbers::inv_sqrtpi_v<float>); // 0.56419

// Natural logarithm of 2
DebugLog(std::numbers::ln2_v<float>); // 0.693147
```

```cpp
// Natural logarithm of 10
DebugLog(std::numbers::ln10_v<float>); // 2.30259

// Square root of 2
DebugLog(std::numbers::sqrt2_v<float>); // 1.41421

// Square root of 3
DebugLog(std::numbers::sqrt3_v<float>); // 1.73205

// 1 divided by the square root of 3
DebugLog(std::numbers::inv_sqrt3_v<float>); // 0.57735

// The Euler-Mascheroni constant
DebugLog(std::numbers::egamma_v<float>); // 0.577216

// The golden ratio
DebugLog(std::numbers::phi_v<float>); // 1.61803
```

For convenience, and as in C#, versions with simplified naming are provided for `double`:

```cpp
DebugLog(std::numbers::pi); // 3.14159
```

Numeric

We'll cover the `<numeric>` header in two parts because it serves two quite different purposes. In this chapter we'll just look at three common numeric algorithms it provides. These aren't available in C#:

```
#include <numeric>

// Greatest common divisor
DebugLog(std::gcd(12, 9)); // 3

// Least common multiple
DebugLog(std::lcm(12, 9)); // 36

// Half way between two numbers
DebugLog(std::midpoint(12.0, 9.0)); // 10.5
```

We'll see the rest of the `<numeric>` header, which deals with sequences of numbers, later in the book when we look at generic algorithms.

Ratio

The `<ratio>` header provides a single class template: `std::ratio`. It takes two integer template parameters representing a numerator and a denominator. It has only two members, `num` and `den`, and both are static. These are calculated at compile time by dividing the template parameters by their greatest common divisor:

```
#include <ratio>

// Greatest common divisor of 1000 and 60 is 20
using MsPerFrame = std::ratio<1000, 60>;

// num = 1000 / 20 = 50
// den = 60 / 20 = 3
DebugLog(MsPerFrame::num, MsPerFrame::den); // 50, 3
```

A bunch of SI ratios are provided to represent powers of 10. Here are a few of them:

```
#include <ratio>

DebugLog(std::nano::num, std::nano::den); // 1,
1000000000
DebugLog(std::milli::num, std::milli::den); // 1, 1000
DebugLog(std::kilo::num, std::kilo::den); // 1000, 1
DebugLog(std::mega::num, std::mega::den); // 1000000, 1
```

The durations we saw in the `<chrono>` header are actually instantiations of `std::ratio`. For example:

Alias	Ratio
`std::chrono::seconds`	`std::ratio<1, 1>`
`std::chrono::minutes`	`std::ratio<60, 1>`
`std::chrono::hours`	`std::ratio<3600, 1>`
`std::chrono::days`	`std::ratio<86400, 1>`

As C# lacks support for integer type parameters to its generic structs and classes, there's no equivalent to this.

Complex

Both languages have support for complex numbers. C# has the `System.Numerics.Complex` struct and C++ has the `std::complex` class template in `<complex>`. That class template has specializations for at least `float`, `double`, and `long double` while the C# version supports only `double`.

Here's how to use `std::complex`:

```
#include <complex>

// Real part is 2. Imaginary part is 0.
std::complex<float> c1{ 2, 0 };
DebugLog(c1.real(), c1.imag()); // 2, 0

// Real part is 0. Imaginary part is 1.
std::complex<float> c2{ 0, 1 };

// Some operators are overloaded
DebugLog(c1 + c2); // 2, 1
DebugLog(c1 - c2); // 2, -1
DebugLog(c1 == c2); // false
DebugLog(c1 != c2); // true
DebugLog(-c1); // -2, -0

// Trigonometric functions
DebugLog(std::sin(c1)); // 0.909297, -0
DebugLog(std::cos(c1)); // -0.416147,-0
```

```
// Hyperbolic functions
DebugLog(std::sinh(c1)); // 3.62686, 0
DebugLog(std::cosh(c1)); // 3.7622, 0

// Exponential functions
DebugLog(std::pow(c1, c2)); // 0.769239, 0.638961
DebugLog(std::sqrt(c1)); // 1.41421, 0

// Misc functions
DebugLog(std::abs(c1)); // 2
DebugLog(std::norm(c1)); // 4
DebugLog(std::conj(c1)); // 2, -0
```

The above is just a sampling of the `std::complex` functionality. Like the C# `Complex` type, quite a bit more is available. C++ also provides user-defined literals in the `std::literals::complex_literals` namespace to create complex numbers with `0` for the real part:

```
#include <complex>

using namespace std::literals::complex_literals;

std::complex<double> d = 2i;
DebugLog(d); // 0, 2

std::complex<float> f = 2if;
DebugLog(f); // 0, 2
```

```
std::complex<long double> ld = 2il;
DebugLog(ld); // 0, 2
```

Bit

The `<bit>` header, introduced in C++20, provides one enumeration for dealing with endianness. This can be used like the `BitConverter.IsLittleEndian` constant in C#:

```
#include <bit>

bool isLittleEndian = std::endian::native ==
std::endian::little;
DebugLog(isLittleEndian); // Maybe true
```

Mainly, this header has functions for performing bit manipulation on integer types:

```
#include <bit>

// Check if only one bit is set, i.e. value is a power of
two
DebugLog(std::has_single_bit(2u)); // true
DebugLog(std::has_single_bit(3u)); // false

// Get the largest power of two greater than or equal to
a value
DebugLog(std::bit_ceil(100u)); // 128

// Rotate bits left, wrapping around
DebugLog(
```

```
    std::rotl(0b10100000000000000000000000000000, 2)
          == 0b10000000000000000000000000000010); //
true

// Count consecutive zero bits starting at the least-
significant
DebugLog(std::countr_zero(0b1000u)); // 3

// Count total one bits
DebugLog(std::popcount(0b10101010101010101010101010101010
)); // 16

// Reinterpret the bits of one type as another type
// Not a real cast, just a function with "cast" in the
name
uint32_t i = std::bit_cast<uint32_t>(3.14f);
DebugLog(i); // 1078523331
```

C# doesn't provide any of these functions, so the closest equivalent would be our own custom implementations of them.

Random

The final numeric header for this chapter is perhaps the most interesting: `<random>`. Like the `Random` class in C#, this header provides functionality for generating random numbers. It is, however, far more advanced than its C# counterpart. For starters, multiple "engines" are available as opposed to the single algorithm that `Random` uses in C#. Here's one of them:

```
#include <random>

// "Subtract with carry" algorithm for uint32_t values
with parameters
std::subtract_with_carry_engine<uint32_t, 24, 10, 24>
swc{};

// Generate random numbers
DebugLog(swc()); // Maybe 15039276
DebugLog(swc()); // Maybe 16323925
DebugLog(swc()); // Maybe 14283486

// Advance the engine state 100 steps without getting any
numbers
swc.discard(100);

// Reset the seed
swc.seed(123);

// Some "subtract with carry" engines with common types
```

```
and parameters
std::ranlux24_base r24; // 32-bit
std::ranlux48_base r48; // 64-bit
```

There are two more available:

```
#include <random>

// "Mersenne Twister" engines
std::mersenne_twister_engine<
    uint32_t, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13>
mt{}; // Custom
std::mt19937 mt32{}; // 32-bit with common parameters
std::mt19937_64 mt64{}; // 64-bit with common parameters

// "Linear congruential generator" engines
std::linear_congruential_engine<uint32_t, 1, 2, 3> lce{};
// Custom
std::minstd_rand0 msr0; // 32-bit "Minimal standard"
std::minstd_rand msr1; // New version of 32-bit "Minimal
standard"
```

There are also some “adapter” engines. These use an underlying engine rather than generating their own random numbers:

```
#include <random>

// std::mt19937 is the underlying engine
```

```
// For each block of 32 random numbers, keep 2 of them
std::discard_block_engine<std::mt19937, 32, 2> db{};
uint32_t dbr = db();

// std::mt19937_64 is the underlying engine generating
64-bit numbers
// Convert them to 32-bit uint32_t values
std::independent_bits_engine<std::mt19937_64, 32,
uint32_t> ib{};
uint32_t ibr = ib();

// std::mt19937 is the underlying engine
// Keep a table of 16 random numbers and shuffle the
order returned
std::shuffle_order_engine<std::mt19937, 16> so{};
uint32_t sor = so();

// Alias of std::shuffle_order_engine<std::minstd_rand0,
256>
std::knuth_b kb{};
```

A `std::random_device` class is also available to provide non-deterministic random numbers on systems that have hardware to produce these. If no hardware is available, a platform-dependent pseudo-random number generator is used instead:

```
#include <random>
```

```
std::random_device rd{};
DebugLog(rd()); // Maybe 448041643
DebugLog(rd()); // Maybe 1317373389
DebugLog(rd()); // Maybe 393151656
```

None of these are typically used directly. That's because they return numbers on their full range of values. We usually want to generate random numbers on some particular range, so we use one of many "distribution" classes. These classes can also shape the random numbers to fit certain patterns:

```
#include <random>

// Random number generator engine
std::mt19937 engine{};

// Normal/Gaussian distribution of float values
// The mean is 3 and the standard deviation is 1.5
std::normal_distribution<float> distribution{ 3.0f, 1.5f
};

// Generate random numbers with the engine on the
distribution
DebugLog(distribution(engine)); // Maybe 3.37974
DebugLog(distribution(engine)); // Maybe 2.56017
DebugLog(distribution(engine)); // Maybe 3.12689
```

20 distribution classes are available to suit a wide variety of needs. Here are a few of them:

```cpp
#include <random>

// Uniform distribution of float values between -1 and 1
std::uniform_real_distribution<float> urd{ -1, 1 };

// Distribution of bool values returning true 75% of the time
std::bernoulli_distribution bd{ 0.75f };

// Gamma distribution of float values with alpha and beta of 1
std::gamma_distribution<float> gd{ 1.0f, 1.0f };

// Distribution of int32_t values that are 0, 1, 2, or 3
// With weights of 3.1, 2.2, 1.6, and 3.4, respectively
std::discrete_distribution<int32_t> dd{3.1f, 2.2f, 1.6f, 3.4f};
```

Conclusion

C++ has a full-featured numerics library. At its most basic there are typed number constants in `<limits>` and `<numbers>` that expand on C# functionality like `int.MaxValue` by adding more constants and fleshing out the offerings so they're available for every type.

The `<numeric>` and `<bit>` headers provide common functions relating to numbers. We can compute the Greatest Common Denominator or the number of ones in an integer. Basic implementations may be easy to write, but the Standard Library implementations are robust, well-tested, optimized, and standardized.

In `<complex>` and `<ratio>` we find some class types to help us work with pairs of numbers, be they real and imaginary or numerator and denominator. In the case of `std::complex`, we get similar functionality as the C# `Complex` type but templates enable support for `float` and `long double` in addition to just `double`. With `std::ratio` we have an easy way to represent ratios like `kilo` and `seconds` at compile time and use them to generate safer, more efficient number conversions.

Finally, there's `<random>` and its suite of random number generation tools. Not only do we get a single algorithm with a few basic tools, as in C#'s `Random` class, but also a full suite of customizable engines, distributions, and even access to hardware-based random number generators.

42. Numbers Library

Limits

C# primitive type structs have `const` fields indicating their range: `int.MinValue` and `int.MaxValue`. Likewise, the C++ Standard Library's `<limits>` header provides the `std::numeric_limits` class template. At its core, this provides a type-safe version of the macros in the C Standard Library's `<limits.h>`/`<climits>` and `<stdint.h>`/`<cstdint>`:

```
#include <limits>

DebugLog(std::numeric_limits<int32_t>::min()); //
-2147483648
DebugLog(std::numeric_limits<int32_t>::max()); //
2147483647
```

The `min` and `max` member functions are `constexpr`, so they can be used in compile-time programming just like the equivalent C# `const` fields.

There are a ton more functions and constants available in `numeric_limits`. Here's a selection of them:

```
#include <limits>

// Difference between 1.0 and the next representable
floating-point value
```

```
DebugLog(std::numeric_limits<float>::epsilon()); //
1.19209e-07

// Largest error in rounding a floating-point value
DebugLog(std::numeric_limits<float>::round_error()); //
0.5

// Floating-point constants
DebugLog(std::numeric_limits<float>::infinity()); // inf
DebugLog(std::numeric_limits<float>::quiet_NaN()); // nan
DebugLog(std::numeric_limits<float>::signaling_NaN()); //
nan

// Type info useful when writing templates
DebugLog(std::numeric_limits<float>::is_integer); //
false
DebugLog(std::numeric_limits<float>::is_exact); // false
DebugLog(std::numeric_limits<float>::is_modulo); // false
DebugLog(std::numeric_limits<float>::digits10); // 6
```

Numbers

The `<numbers>` header was introduced in C++20 to provide mathematical constants in the `std::numbers` namespace. C# has a few of these as `const` fields of `Math`, but the selection is limited and only `double` values are provided. C++ provides a more robust set as variable templates for each numeric type:

```
#include <numbers>

// Base 2 log of e
DebugLog(std::numbers::log2e_v<float>); // 1.4427

// Base 10 log of e
DebugLog(std::numbers::log10e_v<float>); // 0.434294

// Pi
DebugLog(std::numbers::pi_v<float>); // 3.14159

// 1 divided by pi
DebugLog(std::numbers::inv_pi_v<float>); // 0.31831

// 1 divided by the square root of pi
DebugLog(std::numbers::inv_sqrtpi_v<float>); // 0.56419

// Natural logarithm of 2
DebugLog(std::numbers::ln2_v<float>); // 0.693147
```

```cpp
// Natural logarithm of 10
DebugLog(std::numbers::ln10_v<float>); // 2.30259

// Square root of 2
DebugLog(std::numbers::sqrt2_v<float>); // 1.41421

// Square root of 3
DebugLog(std::numbers::sqrt3_v<float>); // 1.73205

// 1 divided by the square root of 3
DebugLog(std::numbers::inv_sqrt3_v<float>); // 0.57735

// The Euler-Mascheroni constant
DebugLog(std::numbers::egamma_v<float>); // 0.577216

// The golden ratio
DebugLog(std::numbers::phi_v<float>); // 1.61803
```

For convenience, and as in C#, versions with simplified naming are provided for `double`:

```cpp
DebugLog(std::numbers::pi); // 3.14159
```

Numeric

We'll cover the `<numeric>` header in two parts because it serves two quite different purposes. In this chapter we'll just look at three common numeric algorithms it provides. These aren't available in C#:

```
#include <numeric>

// Greatest common divisor
DebugLog(std::gcd(12, 9)); // 3

// Least common multiple
DebugLog(std::lcm(12, 9)); // 36

// Half way between two numbers
DebugLog(std::midpoint(12.0, 9.0)); // 10.5
```

We'll see the rest of the `<numeric>` header, which deals with sequences of numbers, later in the book when we look at generic algorithms.

Ratio

The `<ratio>` header provides a single class template: `std::ratio`. It takes two integer template parameters representing a numerator and a denominator. It has only two members, `num` and `den`, and both are static. These are calculated at compile time by dividing the template parameters by their greatest common divisor:

```
#include <ratio>

// Greatest common divisor of 1000 and 60 is 20
using MsPerFrame = std::ratio<1000, 60>;

// num = 1000 / 20 = 50
// den = 60 / 20 = 3
DebugLog(MsPerFrame::num, MsPerFrame::den); // 50, 3
```

A bunch of SI ratios are provided to represent powers of 10. Here are a few of them:

```
#include <ratio>

DebugLog(std::nano::num, std::nano::den); // 1,
1000000000
DebugLog(std::milli::num, std::milli::den); // 1, 1000
DebugLog(std::kilo::num, std::kilo::den); // 1000, 1
DebugLog(std::mega::num, std::mega::den); // 1000000, 1
```

The durations we saw in the `<chrono>` header are actually instantiations of `std::ratio`. For example:

Alias	Ratio
`std::chrono::seconds`	`std::ratio<1, 1>`
`std::chrono::minutes`	`std::ratio<60, 1>`
`std::chrono::hours`	`std::ratio<3600, 1>`
`std::chrono::days`	`std::ratio<86400, 1>`

As C# lacks support for integer type parameters to its generic structs and classes, there's no equivalent to this.

Complex

Both languages have support for complex numbers. C# has the `System.Numerics.Complex` struct and C++ has the `std::complex` class template in `<complex>`. That class template has specializations for at least `float`, `double`, and `long double` while the C# version supports only `double`.

Here's how to use `std::complex`:

```
#include <complex>

// Real part is 2. Imaginary part is 0.
std::complex<float> c1{ 2, 0 };
DebugLog(c1.real(), c1.imag()); // 2, 0

// Real part is 0. Imaginary part is 1.
std::complex<float> c2{ 0, 1 };

// Some operators are overloaded
DebugLog(c1 + c2); // 2, 1
DebugLog(c1 - c2); // 2, -1
DebugLog(c1 == c2); // false
DebugLog(c1 != c2); // true
DebugLog(-c1); // -2, -0

// Trigonometric functions
DebugLog(std::sin(c1)); // 0.909297, -0
DebugLog(std::cos(c1)); // -0.416147,-0
```

```
// Hyperbolic functions
DebugLog(std::sinh(c1)); // 3.62686, 0
DebugLog(std::cosh(c1)); // 3.7622, 0

// Exponential functions
DebugLog(std::pow(c1, c2)); // 0.769239, 0.638961
DebugLog(std::sqrt(c1)); // 1.41421, 0

// Misc functions
DebugLog(std::abs(c1)); // 2
DebugLog(std::norm(c1)); // 4
DebugLog(std::conj(c1)); // 2, -0
```

The above is just a sampling of the `std::complex` functionality. Like the C# `Complex` type, quite a bit more is available. C++ also provides user-defined literals in the `std::literals::complex_literals` namespace to create complex numbers with `0` for the real part:

```
#include <complex>

using namespace std::literals::complex_literals;

std::complex<double> d = 2i;
DebugLog(d); // 0, 2

std::complex<float> f = 2if;
DebugLog(f); // 0, 2
```

```
std::complex<long double> ld = 2il;
DebugLog(ld); // 0, 2
```

Bit

The `<bit>` header, introduced in C++20, provides one enumeration for dealing with endianness. This can be used like the `BitConverter.IsLittleEndian` constant in C#:

```
#include <bit>

bool isLittleEndian = std::endian::native ==
std::endian::little;
DebugLog(isLittleEndian); // Maybe true
```

Mainly, this header has functions for performing bit manipulation on integer types:

```
#include <bit>

// Check if only one bit is set, i.e. value is a power of
two
DebugLog(std::has_single_bit(2u)); // true
DebugLog(std::has_single_bit(3u)); // false

// Get the largest power of two greater than or equal to
a value
DebugLog(std::bit_ceil(100u)); // 128

// Rotate bits left, wrapping around
DebugLog(
```

```
    std::rotl(0b10100000000000000000000000000000, 2)
          == 0b10000000000000000000000000000010); //
true

// Count consecutive zero bits starting at the least-
significant
DebugLog(std::countr_zero(0b1000u)); // 3

// Count total one bits
DebugLog(std::popcount(0b10101010101010101010101010101010
)); // 16

// Reinterpret the bits of one type as another type
// Not a real cast, just a function with "cast" in the
name
uint32_t i = std::bit_cast<uint32_t>(3.14f);
DebugLog(i); // 1078523331
```

C# doesn't provide any of these functions, so the closest equivalent would be our own custom implementations of them.

Random

The final numeric header for this chapter is perhaps the most interesting: `<random>`. Like the `Random` class in C#, this header provides functionality for generating random numbers. It is, however, far more advanced than its C# counterpart. For starters, multiple "engines" are available as opposed to the single algorithm that `Random` uses in C#. Here's one of them:

```
#include <random>

// "Subtract with carry" algorithm for uint32_t values
with parameters
std::subtract_with_carry_engine<uint32_t, 24, 10, 24>
swc{};

// Generate random numbers
DebugLog(swc()); // Maybe 15039276
DebugLog(swc()); // Maybe 16323925
DebugLog(swc()); // Maybe 14283486

// Advance the engine state 100 steps without getting any
numbers
swc.discard(100);

// Reset the seed
swc.seed(123);

// Some "subtract with carry" engines with common types
```

```
and parameters
std::ranlux24_base r24; // 32-bit
std::ranlux48_base r48; // 64-bit
```

There are two more available:

```
#include <random>

// "Mersenne Twister" engines
std::mersenne_twister_engine<
    uint32_t, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13>
mt{}; // Custom
std::mt19937 mt32{}; // 32-bit with common parameters
std::mt19937_64 mt64{}; // 64-bit with common parameters

// "Linear congruential generator" engines
std::linear_congruential_engine<uint32_t, 1, 2, 3> lce{};
// Custom
std::minstd_rand0 msr0; // 32-bit "Minimal standard"
std::minstd_rand msr1; // New version of 32-bit "Minimal
standard"
```

There are also some “adapter” engines. These use an underlying engine rather than generating their own random numbers:

```
#include <random>

// std::mt19937 is the underlying engine
```

```
// For each block of 32 random numbers, keep 2 of them
std::discard_block_engine<std::mt19937, 32, 2> db{};
uint32_t dbr = db();

// std::mt19937_64 is the underlying engine generating
64-bit numbers
// Convert them to 32-bit uint32_t values
std::independent_bits_engine<std::mt19937_64, 32,
uint32_t> ib{};
uint32_t ibr = ib();

// std::mt19937 is the underlying engine
// Keep a table of 16 random numbers and shuffle the
order returned
std::shuffle_order_engine<std::mt19937, 16> so{};
uint32_t sor = so();

// Alias of std::shuffle_order_engine<std::minstd_rand0,
256>
std::knuth_b kb{};
```

A `std::random_device` class is also available to provide non-deterministic random numbers on systems that have hardware to produce these. If no hardware is available, a platform-dependent pseudo-random number generator is used instead:

```
#include <random>
```

```
std::random_device rd{};
DebugLog(rd()); // Maybe 448041643
DebugLog(rd()); // Maybe 1317373389
DebugLog(rd()); // Maybe 393151656
```

None of these are typically used directly. That's because they return numbers on their full range of values. We usually want to generate random numbers on some particular range, so we use one of many "distribution" classes. These classes can also shape the random numbers to fit certain patterns:

```
#include <random>

// Random number generator engine
std::mt19937 engine{};

// Normal/Gaussian distribution of float values
// The mean is 3 and the standard deviation is 1.5
std::normal_distribution<float> distribution{ 3.0f, 1.5f
};

// Generate random numbers with the engine on the
distribution
DebugLog(distribution(engine)); // Maybe 3.37974
DebugLog(distribution(engine)); // Maybe 2.56017
DebugLog(distribution(engine)); // Maybe 3.12689
```

20 distribution classes are available to suit a wide variety of needs. Here are a few of them:

```cpp
#include <random>

// Uniform distribution of float values between -1 and 1
std::uniform_real_distribution<float> urd{ -1, 1 };

// Distribution of bool values returning true 75% of the time
std::bernoulli_distribution bd{ 0.75f };

// Gamma distribution of float values with alpha and beta of 1
std::gamma_distribution<float> gd{ 1.0f, 1.0f };

// Distribution of int32_t values that are 0, 1, 2, or 3
// With weights of 3.1, 2.2, 1.6, and 3.4, respectively
std::discrete_distribution<int32_t> dd{3.1f, 2.2f, 1.6f, 3.4f};
```

Conclusion

C++ has a full-featured numerics library. At its most basic there are typed number constants in `<limits>` and `<numbers>` that expand on C# functionality like `int.MaxValue` by adding more constants and fleshing out the offerings so they're available for every type.

The `<numeric>` and `<bit>` headers provide common functions relating to numbers. We can compute the Greatest Common Denominator or the number of ones in an integer. Basic implementations may be easy to write, but the Standard Library implementations are robust, well-tested, optimized, and standardized.

In `<complex>` and `<ratio>` we find some class types to help us work with pairs of numbers, be they real and imaginary or numerator and denominator. In the case of `std::complex`, we get similar functionality as the C# `Complex` type but templates enable support for `float` and `long double` in addition to just `double`. With `std::ratio` we have an easy way to represent ratios like `kilo` and `seconds` at compile time and use them to generate safer, more efficient number conversions.

Finally, there's `<random>` and its suite of random number generation tools. Not only do we get a single algorithm with a few basic tools, as in C#'s `Random` class, but also a full suite of customizable engines, distributions, and even access to hardware-based random number generators.

43. Threading Library

Thread

Like the C# `Thread` class in `System.Threading`, C++ has `std::thread` in the `<thread>` header. It's the most basic way to create another thread:

```
#include <thread>
#include <chrono>

void PrintLoop(const char* threadName)
{
    for (int i = 0; i < 10; ++i)
    {
        DebugLog(threadName, i);

        // this_thread provides functions that operate on
the current thread
        // sleep_for takes a std::chrono::duration

std::this_thread::sleep_for(std::chrono::milliseconds{100
});
    }
}

void MyThread()
{
```

```
    PrintLoop("MyThread");
}

// Create a thread and immediately start executing
MyThread in it
std::thread t{MyThread};

// This happens on the main thread
PrintLoop("Main Thread");

// Block until the thread terminates
t.join();
```

This prints something like this, depending on OS thread scheduling:

```
Main Thread, 0
MyThread, 0
MyThread, 1
Main Thread, 1
Main Thread, 2
MyThread, 2
MyThread, 3
Main Thread, 3
Main Thread, 4
MyThread, 4
MyThread, 5
Main Thread, 5
MyThread, 6
```

```
Main Thread, 6
MyThread, 7
Main Thread, 7
Main Thread, 8
MyThread, 8
Main Thread, 9
MyThread, 9
```

`std::this_thread` has a few other functions:

```
#include <thread>

// Sleep until a specific time
std::this_thread::sleep_until(std::chrono::system_clock::
now() + 1500ms);

// Tell the OS to schedule other threads
std::this_thread::yield();

// Get the current thread's ID
// Has overloaded comparison operators and works with
std::hash
std::thread::id i = std::this_thread::get_id();
std::thread t{
    [&] {
        DebugLog(i == std::this_thread::get_id()); //
false
    }
```

```
};
t.join();
```

`std::thread` itself has just a little more functionality. For starters, we can pass parameters to threads:

```
#include <thread>

void Thread(int param)
{
    DebugLog(param); // 123 then 456 or visa versa
}

std::thread t1{ Thread, 123 };
std::thread t2{ Thread, 456 };
t1.join();
t2.join();
```

We can get some information about the thread, including its `std::thread::id`:

```
#include <thread>

std::thread t{ [] {} };

// Get the ID from outside the thread
std::thread::id id = t.get_id();
```

```
// Get an platform-dependent handle to the thread
std::thread::native_handle_type handle =
t.native_handle();

// Check how many threads the CPU can run at once
// Depends on number of processors, cores, Hyper-
threading, etc.
unsigned int hc = std::thread::hardware_concurrency();
DebugLog(hc); // Maybe 8

// Check if the thread is active, i.e. we can join() it
DebugLog(t.joinable()); // true

t.join();

DebugLog(t.joinable()); // false
```

The last function is `detach`, which releases the OS thread from the `std::thread`:

```
#include <thread>
#include <chrono>

std::thread t{ [] {
    DebugLog("thread start");

std::this_thread::sleep_for(std::chrono::milliseconds{500
});
```

```
    DebugLog("thread done");
} };

// Release the OS thread
t.detach();

DebugLog(t.joinable()); // false

DebugLog("main thread done");

// Can't join() the thread anymore, so sleep longer than
it runs
std::this_thread::sleep_for(std::chrono::milliseconds{
1000 });
```

This might print:

```
false
main thread done
thread start
thread done
```

The reason this is necessary is that the `std::thread` destructor throws an exception if the thread is joinable. Calling `detach` makes it non-joinable:

```
#include <thread>
#include <chrono>
```

```
void Foo()
{
    std::thread t{ [] {

std::this_thread::sleep_for(std::chrono::milliseconds{500
});
    } };
} // destructor throws
```

Next in `<thread>`, which debuted in C++20, is `std::jthread`. This is like `std::thread` but with support for cancelation and automatic joining. The `std::jthread` destructor calls `join` for us if the thread is joinable. As C# lacks destructors, there's no equivalent to this:

```
#include <thread>
#include <chrono>

void Foo()
{
    std::jthread t{ [] {

std::this_thread::sleep_for(std::chrono::milliseconds{500
});
    } };
} // destructor calls join()
```

To support cancelation, the thread function can take a `std::stop_token` defined in `<stop_token>` used to check if another thread has requested that the thread stop executing. Using this "stop" functionality allows us to avoid some tricky inter-thread communication. Unfortunately, there's no analog to this in C#:

```
#include <thread>
#include <chrono>

void Foo()
{
    // Thread function takes a stop_token
    std::jthread t{ [] (std::stop_token st) {
        // Check if a stop is requested
        while (!st.stop_requested())
        {

std::this_thread::sleep_for(std::chrono::milliseconds{100
});
            DebugLog("Thread still running");
        }

        DebugLog("Stop requested");

std::this_thread::sleep_for(std::chrono::milliseconds{
500 });
    } };

    std::this_thread::sleep_for(std::chrono::seconds{ 1
```

```
});

    // Request that the thread stop executing
    // This does not block like join() would
    t.request_stop();
    DebugLog("After requesting stop");
} // jthread destructor calls join(). About 500
milliseconds passes here...
```

Stop Token

Besides defining `std::stop_token`, the `<stop_token>` header has a couple other features related to `std::jthread`. First, there is `std::stop_source` which issues `std::stop_token` objects:

```
#include <stop_token>

// Create a source of tokens
std::stop_source source{};

// Issue tokens from the source
std::stop_token t1 = source.get_token();
std::stop_token t2 = source.get_token();

// No stop is initially requested
DebugLog(t1.stop_requested(), t2.stop_requested()); //
false, false

// Request a stop on all tokens issued by the source
source.request_stop();
DebugLog(t1.stop_requested(), t2.stop_requested()); //
true, true
```

A `std::stop_callback` allows for a function to be called when a stop is requested:

```
#include <stop_token>

std::stop_source source{};
std::stop_token t = source.get_token();

// Call a lambda when a token's source is stopped
std::stop_callback sc(
    t,
    [] { DebugLog("stop requested"); });

source.request_stop(); // stop requested
```

The callback is called on the thread that requests the stop:

```
#include <thread>
#include <stop_token>
#include <chrono>

// Thread that sleeps for 1 second
std::jthread t1{
    []
    {

std::this_thread::sleep_for(std::chrono::milliseconds{100
0});
    }
};
```

```
std::stop_source source1 = t1.get_stop_source();
std::stop_token token1 = t1.get_stop_token();

// Thread that sleeps for 0.5 seconds then stops thread
1's source
std::jthread t2{
    [](std::stop_source ss)
    {

std::this_thread::sleep_for(std::chrono::milliseconds{500
});

        // Calls the below callback on this thread
        ss.request_stop();
    },
    source1
};
std::thread::id id2 = t2.get_id();

// Register a callback for when thread 1's token is
stopped
std::stop_callback sc{
    token1,
    [&]
    {
        // Print which thread the callback was called on
        DebugLog(id2 == std::this_thread::get_id()); //
true
```

```
    }
};

// Wait 2 seconds for the threads to do their work
std::this_thread::sleep_for(std::chrono::milliseconds{
2000 });
```

Mutex

Proper synchronization between threads is essential to prevent data corruption and logic errors. To this end, C++ provides numerous facilities starting with `std::mutex`, the equivalent of C#'s `Mutex` class.

```
#include <thread>
#include <mutex>

// An array to fill up with integers
constexpr int size = 10;
int integers[size];
int index = 0;

// A mutex to control access to the array
std::mutex m{};

auto writer = [&]
{
    while (true)
    {
        // Lock the mutex before accessing shared state:
index and integers
        m.lock();

        // Access shared state by reading index
        if (index >= size)
        {
```

```
            // Unlock the mutex when done with the shared
state
            m.unlock();
            break;
        }

        // Access shared state by reading and writing
index and writing integers
        integers[index] = index;
        index++;

        // Unlock the mutex when done with the shared
state
        m.unlock();
    }
};

std::thread t1{writer};
std::thread t2{writer};

t1.join();
t2.join();

for (int i = 0; i < size; ++i)
{
    DebugLog(integers[i]); // 0, 1, 2, 3, 4, 5, 6, 7, 8,
9
}
```

More mutex classes are available besides the basic `std::mutex`. The `std::timed_mutex` class allows us to attempt to unlock a mutex for a certain amount of time:

```
#include <mutex>

std::timed_mutex m{};

// Try to get a lock for up to 1 millisecond then give up
bool didLock = m.try_lock_for(std::chrono::milliseconds{
1 });
```

Because `std::mutex` and `std::timed_mutex` can't be locked when already locked by the same thread, there's also `std::recursive_mutex` that allows for this:

```
#include <mutex>

std::recursive_mutex m{};

// First lock
m.lock();

// Second lock: OK with recursive_mutex but not regular
mutex
m.lock();

// Unlock second lock
```

```
m.unlock();

// Unlock first lock
m.unlock();
```

The `std::recursive_timed_mutex` class combines `std::recursive_mutex` and `std::timed_mutex` to provide both their feature sets.

When multiple mutexes need to be locked, a `std::lock` function avoids deadlocks due to the ABA problem:

```
std::mutex m1{};
std::mutex m2{};

// Lock both mutexes
std::lock(m1, m2);

m1.unlock();
m2.unlock();
```

When we need to call a function exactly once from multiple threads, we can use `std::call_once` and its helper class `std::once_flag`:

```
#include <thread>
#include <mutex>

// Keeps track of whether the function has been called
std::once_flag of{};
```

```
// Function to call once
auto print = [](int x) { DebugLog("called once", x); };

// Two threads racing to call the function
auto threadFunc = [&](int x) { std::call_once(of, print,
x); };
std::thread t1{ threadFunc, 123 };
std::thread t2{ threadFunc, 456 };

t1.join();
t2.join();
```

In C#, we rarely use `Mutex` directly. Instead, we usually prefer to use a `lock` statement which takes care of unlocking the mutex even when an exception is thrown. The same is true in C++, except that we use a lock class and its destructor unlocks the mutex even when an exception is thrown:

```
#include <mutex>

void Foo()
{
    // Mutex to lock
    std::mutex m{};

    // Create a lock object for the mutex
    // Constructor locks the mutex
```

```
    std::lock_guard g{ m };
} // lock_guard's destructor unlocks the mutex
```

`std::unique_lock` is the same as `std::lock_guard` but it supports "moving" the lock object and not "copying" it. Regardless of the lock class chosen, we can use `std::lock` to lock multiple if we first defer taking the lock:

```
#include <mutex>

void Foo()
{
    std::mutex m1{};
    std::mutex m2{};

    // Make the locks, but don't lock the mutexes yet
    std::unique_lock g1{ m1, std::defer_lock };
    std::unique_lock g2{ m2, std::defer_lock };

    // Lock them both, avoiding deadlocks
    std::lock(g1, g2);
} // unique_lock's destructor unlocks both mutexes
```

In C++17, a more convenient `std::scoped_lock` was added to deal with locking multiple mutexes:

```
#include <mutex>
```

```
void Foo()
{
    std::mutex m1{};
    std::mutex m2{};

    // Lock both mutexes
    std::scoped_lock g{ m1, m2 };
} // scoped_lock's destructor unlocks both mutexes
```

Shared Mutex

C++17 adds another mutex type, `std::shared_mutex`, in the `<shared_mutex>` header. There are two ways to lock this mutex: “exclusive” and “shared.” An “exclusive” lock can only be taken by one thread at a time and prevents any threads from taking a “shared” lock. A “shared” lock allows other threads to take a “shared” lock but not an “exclusive” lock. Regardless of the kind of lock, any given thread can only lock once.

To take these two kinds of locks, we use the `std::unique_lock` class we’ve already seen in `<mutex>` and the `std::shared_lock` or `std::shared_timed_lock` classes provided by `<shared_mutex>`:

```
#include <mutex>
#include <shared_mutex>

class SharedInt
{
    int Value = 0;

    // Mutex that protects the value
    mutable std::shared_mutex Mutex;

public:

    int GetValue() const
    {
        // Multiple threads can read at once, so use take
a "shared" lock
```

```
        std::shared_lock lock{ Mutex };

        return Value;
    }

    void SetValue(int value)
    {
        // Only one thread can write at once, so take an
"exclusive" lock
        std::unique_lock lock{ Mutex };
        Value = value;
    }
};
```

Semaphore

C++20 introduces more synchronization mechanisms than just mutexes, starting with `std::counting_semaphore` in `<semaphore>`. This is the analog of C#'s `Semaphore` class and it allows more than one access at a time:

```
#include <semaphore>

// Allow up to 3 accesses with the counter starting at 3
std::counting_semaphore<3> cs{ 3 };

// Block while the counter is 0 then decrement it by 1
cs.acquire();
// Counter is now 2

cs.acquire();
// Counter is now 1

cs.acquire();
// Counter is now 0

// Try to acquire, but fail because the counter is at 0
bool didAcquire = cs.try_acquire();
DebugLog(didAcquire); // false

// Increment the counter
cs.release();
```

```
// Counter is now 1

DebugLog(cs.try_acquire()); // true
// Counter is now 0
```

A `std::binary_semaphore` is provided as an alias of `std::counting_semaphore<1>`.

Barrier

The next synchronization mechanism provided by C++20 is `std::barrier` in the `<barrier>` header. It's equivalent to the `Barrier` class in C#. Like a semaphore, a barrier has a count of threads. In contrast, these indicate threads that have "arrived" at the barrier and should block until the barrier is "completed":

```
#include <chrono>
#include <thread>
#include <barrier>

void Foo()
{
    // Define a function to call when the barrier is
completed
    auto complete = []() noexcept {};

    // Allow up to three threads to block until the
barrier is completed
    std::barrier b{ 3, complete };

    auto threadFunc = [&](int id)
    {
        // Do something before arriving at the barrier
        DebugLog("before arrival", id);

        // Arrive at the barrier and get a token
        auto arrivalToken = b.arrive();
```

```
        // Do something after arriving at the barrier
        DebugLog("after arrival", id);

        // Wait for the barrier to complete
        b.wait(std::move(arrivalToken));

        // Do something after the barrier completes
        DebugLog("after waiting", id);
    };
    std::jthread t1{ threadFunc, 1 };
    std::jthread t2{ threadFunc, 2 };
    std::jthread t3{ threadFunc, 3 };

    std::this_thread::sleep_for(std::chrono::seconds{ 3
});

    // Complete the barrier
    complete();
}
```

This prints:

```
before arrival, 3
before arrival, 1
before arrival, 2
after arrival, 3
```

```
after arrival, 11
after arrival, 2
```

Then three seconds later…

```
after waiting, 2
after waiting, 1
after waiting, 3
```

Latch

The `std::latch` class in C++20's `<latch>` header provides a single-use version of `std::barrier`. This class is flexible in different ways than `std::barrier`. One is that any given thread can decrement the counter more than once. Another is that decrementing can be by more than one step. There's no completion function though. Instead, threads blocking on the latch are resumed when the counter hits zero.

```
#include <chrono>
#include <thread>
#include <latch>

// Allow up to three threads to block
std::latch latch{ 3 };

auto threadFunc = [&](int id)
{
    // Do something before
    DebugLog("before", id);

    // Decrement the counter by one
    latch.count_down();

    // Do something after
    DebugLog("after", id);

    // Wait for the counter to hit zero
```

```
    latch.wait();

    // Do something after the counter hits zero
    DebugLog("after zero", id);
};
std::jthread t1{ threadFunc, 1 };
std::this_thread::sleep_for(std::chrono::seconds{ 2 });
std::jthread t2{ threadFunc, 2 };
std::this_thread::sleep_for(std::chrono::seconds{ 2 });
std::jthread t3{ threadFunc, 3 };
```

This prints:

```
before, 1
after, 1
```

Then two seconds later…

```
before, 2
after, 2
```

And two more seconds later, thread 3 reduces the latch to zero…

```
before, 3
after, 3
after zero, 3
```

```
after zero, 1
after zero, 2
```

There’s no direct equivalent in C#, but `Barrier` and `CountdownEvent` are rather close.

Condition Variable

Another thread synchronization option is `std::condition_variable` in `<condition_variable>`. This is similar to the `ManualResetEvent` and `AutoResetEvent` classes in C# in that it's used by a thread that needs to wait for some condition to be satisfied before proceeding:

```
#include <thread>
#include <mutex>
#include <condition_variable>

// Mutex and condition variable to coordinate the threads
std::mutex m;
std::condition_variable cv;

// Flags to indicate that work is ready and the result of
work is ready
bool workReady;
bool resultReady;

// The result of work
int result;

// Thread that does the work
// First waits for the condition to be set indicating
that work is ready
void WorkThread()
{
```

```
    // Lock the mutex
    DebugLog("Work thread locking mutex");
    std::unique_lock<std::mutex> lock(m);

    // Wait for the workReady flag to be set to true
    DebugLog("Work thread waiting for workReady flag");
    cv.wait(lock, [] { return workReady; });

    // Now we have the mutex locked
    // Do "work" by setting the shared value to 123
    DebugLog("Work thread doing work");
    result = 123;

    // Set the resultReady flag to tell the other thread
our work is done
    DebugLog("Work thread setting resultReady flag");
    resultReady = true;

    // Unlock the mutex
    DebugLog("Work thread unlocking mutex");
    lock.unlock();

    // Notify the condition variable
    DebugLog("Work thread notifying CV");
    cv.notify_one();

    DebugLog("Work thread done");
}
```

```
// Initially nothing is ready
result = 0;
workReady = false;
resultReady = false;

// Start the thread
// It'll start waiting for the condition variable
std::thread worker{ WorkThread };

{
    // Lock the mutex
    DebugLog("Main thread locking mutex");
    std::lock_guard lg(m);

    // Set the flag to indicate that work is ready
    DebugLog("Main thread setting workReady flag");
    workReady = true;
} // Third, unlock the mutex (via lock_guard destructor)

// Fourth, notify the condition variable
DebugLog("Main thread notifying CV");
cv.notify_one();

{
    // Lock the mutex
    DebugLog("Main thread locking mutex to get result");
    std::unique_lock ul(m);
```

```
    // Wait for the resultReady flag to be set to true
    DebugLog("Main thread waiting for resultReady flag");
    cv.wait(ul, [] { return resultReady; });
}

// Use the result
DebugLog("Main thread got result", result);

worker.join();
```

This prints:

```
Main thread locking mutex
Main thread setting workReady flag
Main thread notifying CV
Main thread locking mutex to get result
Main thread waiting for resultReady flag
Work thread locking mutex
Work thread waiting for workReady flag
Work thread doing work
Work thread setting resultReady flag
Work thread unlocking mutex
Work thread notifying CV
Work thread done
Main thread got result, 123
```

`std::condition_variable` requires us to use exactly `std::mutex` as our mutex type. If we'd rather use another type, we can replace it with `std::condition_variable_any`.

Atomic

The final synchronization mechanism we'll look at in this chapter is `std::atomic` in the `<atomic>` header. A `std::atomic<T>` class acts like the `T` type but all operators are implemented atomically. This can be somewhat approximated in C# with the static functions of the `Interlocked` class, but there's no generic type that behaves quite like `std::atomic`.

```
#include <atomic>
#include <thread>

// Make an atomic int starting at zero
std::atomic<int> val{ 0 };

// Run three threads that each use the atomic int
auto threadFunc = [&]
{
    for (int i = 0; i < 1000; ++i)
    {
        // Call the overloaded ++ operator
        // Atomically adds one
        val++;
    }
};
std::jthread t1{ threadFunc };
std::jthread t2{ threadFunc };
std::jthread t3{ threadFunc };
t1.join();
```

```
t2.join();
t3.join();

DebugLog(val); // 3000
```

There are a lot of type aliases for specializations of the `std::atomic` class template. Here are a few:

```
std::atomic_bool ab; // atomic<bool>
std::atomic_int ai; // atomic<int>
std::atomic_int32_t ai32; // atomic<int32_t>
std::atomic_int64_t ai64; // atomic<int64_t>
std::atomic_size_t as; // atomic<size_t>

// C++20: the most efficient lock-free types
std::atomic_signed_lock_free aslf; // signed
std::atomic_unsigned_lock_free aulf; // unsigned
```

Any trivially-copyable, copy-constructible, and copy-assignable type can be used but hardware support for atomic operations may not be available and locks may be required to ensure atomic access:

```
struct Player
{
    const char* Name;
    int32_t Score;
    int32_t Health;
```

```
};
std::atomic<Player> ap;
```

We can test that with `is_lock_free`:

```
std::atomic<int> val{ 0 };
DebugLog(val.is_lock_free()); // true
DebugLog(ap.is_lock_free()); // false
```

Besides overloaded operators like `++`, there are a few member functions to take more control over the atomic operations. The `store` and `load` functions allow customization of how memory is affected so we can control memory reordering:

```
#include <atomic>

std::atomic<int> val{ 0 };

// Write and customize how memory ordering is affected
val.store(1, std::memory_order_relaxed); // No
synchronization
val.store(2, std::memory_order_release); // No writes
reordered after this
val.store(3, std::memory_order_seq_cst); // Sequentially
consistent

// Read and customize how memory ordering is affected
int i;
```

```
i = val.load(std::memory_order_relaxed); // No
synchronization
i = val.load(std::memory_order_consume); // No writes
reordered before this
i = val.load(std::memory_order_acquire); // No reads or
writes before this
i = val.load(std::memory_order_seq_cst); // Sequentially
consistent
```

The `exchange`, `compare_exchange_weak`, and `compare_exchange_strong` functions are very similar to functions in C#'s `Interlocked` class:

```
#include <atomic>

std::atomic<int> v{ 123 };

// Set a new value and return the old value
int old = v.exchange(456);
DebugLog(old); // 123

// Set a new value if the current value is an expected
value
int expected = 456;
bool exchanged = v.compare_exchange_strong(expected,
789);
DebugLog(exchanged); // true
DebugLog(v); // 789
```

```
exchanged = v.compare_exchange_strong(expected, 1000);
DebugLog(exchanged); // false
DebugLog(v); // 789

// A "weak" version that might set even if the expected
value differs
exchanged = v.compare_exchange_strong(expected, 1000);
```

Future

Lastly, we have `<future>` with its `future` and `async` functionality. The `async` function works conceptually similarly to `Task` in C# in that the platform takes care of running a function, presumably on another thread in a thread pool. A `future` is returned as a placeholder for the eventual return value of that function:

```
#include <chrono>
#include <thread>
#include <future>

DebugLog("Calling async");
std::future<int> f {
    std::async(
        [] {

std::this_thread::sleep_for(std::chrono::seconds{2});
            return 123;
        }
    )
};

DebugLog("Waiting");
f.wait();

DebugLog("Getting return value");
```

```
int retVal = f.get();
DebugLog("Got return value", retVal);
```

This prints:

```
Calling async
Waiting
```

Then two seconds later…

```
Getting return value
Got return value, 123
```

The `std::launch` enumeration provides options for how to execute the function. By default, it's run asynchronously as though we passed `std::launch::async`. We can instead use `std::launch::deferred` to instead run the function on the first thread that calls `get` on the `future`:

```
#include <chrono>
#include <thread>
#include <future>

DebugLog("Calling async");
std::future<int> f{
    std::async(std::launch::deferred, [] {

std::this_thread::sleep_for(std::chrono::seconds{2});
```

```
        return 123; }) };

DebugLog("Doing something else");
std::this_thread::sleep_for(std::chrono::seconds{ 5 });

DebugLog("Getting return value");
int retVal = f.get();
DebugLog("Got return value", retVal);
```

This prints:

```
Calling async
Doing something else
```

Then five seconds later calling `get` runs the function…

```
Getting return value
```

Then two more seconds later the function finishes…

```
Got return value, 123
```

`std::promise` is another way to create a `std::future` besides `std::async`. It can also hold an exception in the case that no value can be produced. We can use `std::promise` with a wide variety of asynchronous programming techniques. For example, it easily combines with a simple `std::jthread`:

```
#include <chrono>
#include <thread>
#include <future>

// Make a promise to produce an int
std::promise<int> p{};

// Get a future for that int
std::future<int> f{ p.get_future() };

// Produce the value in another thread
std::jthread{
    [&]
    {

std::this_thread::sleep_for(std::chrono::seconds{2});
        p.set_value(123);
    }
};

// Block until the value is ready
DebugLog("Getting return value");
int retVal = f.get();
DebugLog(retVal);
```

This prints:

```
Getting return value
```

Then two seconds later…

```
123
```

The last class to look at in `<future>` is `std::packaged_task`. This is an adapter that wraps functions in a class with an overloaded function call operator like is done for us by the compiler with lambdas. We can then get a `std::future` that's ready when the `std::packaged_task` is called. Like `std::promise`, we can use `std::packaged_task` with plain threads or other asynchronous programming paradigms besides `std::async`:

```
#include <chrono>
#include <thread>
#include <future>

int DoWork()
{
    std::this_thread::sleep_for(std::chrono::seconds{ 2
});
    return 123;
}

// Wrap DoWork in a class object
std::packaged_task pt{ DoWork };
```

```
// Get a future for when the packaged task is executed
std::future<int> f{ pt.get_future() };

// Call the packaged task on another thread
std::jthread t{ [&] { pt(); } };

// Block (for about 2 seconds) until DoWork returns
DebugLog("Getting return value");
int retVal = f.get();
DebugLog(retVal);
```

This prints:

```
Getting return value
```

Then two seconds later…

```
123
```

Conclusion

The C++ Standard Library provides us with quite a few multi-threading tools. At the most basic, we have `thread` and `jthread` to create our own threads. Once we've created these, we have a huge variety of synchronization mechanisms: mutexes, latches, barriers, semaphores, condition variables, and atomics. The `<future>` header provides `future`, `promise`, `packaged_task` to wrap up work that'll be done asynchronously and either complete or throw an exception in the future. These generic tools allow us to avoid implementing extremely complex and error-prone thread synchronization strategies ourselves.

The Standard Library even provides `async` as a high-level mechanism for letting the platform take care of scheduling threads in a manner similar to C#'s `Task`. A future version of the Standard Library will combine this with coroutines for a very similar experience to C#'s `async` functions. In the meantime, we can make use of community libraries to accomplish this.

44. Strings Library

Charconv

C++17 introduces `<charconv>` with a pair of functions for converting primitive types like `double` to characters and reading them back from characters. These functions don't allocate memory, throw exceptions, handle localization, or even add NUL terminators. They're intended to be used in serialization such as to JSON or when sending strings over a network socket:

```
#include <charconv>

// Buffer to print the value to
char buf[100];
char* end = buf + sizeof(buf);

// Print 3.14 to the buffer in scientific notation
std::to_chars_result tcr{
    std::to_chars(buf, end, 3.14,
std::chars_format::scientific) };

// Add a NUL terminator to the returned pointer to the
character after the
// last printed character
*tcr.ptr = '\0';

DebugLog(buf); // 3.14e+00
DebugLog("Success?", tcr.ec == std::errc()); // true
```

```
DebugLog("End pointer index", tcr.ptr - buf); // 8

// Read 3.14e+00 from the buffer
double val;
std::from_chars_result fcr{ std::from_chars(buf, end,
val) };

DebugLog(val); // 3.14
DebugLog("Success?", fcr.ec == std::errc()); // true
DebugLog("End pointer index", fcr.ptr - buf); // 8
```

The `TextReader` and `TextWriter` classes in C# are probably the closest analog as they can write to existing streams rather than operating on individual `string` objects.

String

Next up is `<string>` which primarily defines the `std::basic_string` class template. This is similar to the built-in `String`/`string` type in C#. One key difference is that it is mutable, meaning that the string's characters can change. It is also not a managed reference, as C++ doesn't have those, and must be wrapped in something like a std::shared_ptr for a similar effect.

A template parameter of `std::basic_string` specifies the type of characters in the string. The `<string>` header provides many aliases for common character types so it's rare to use `std::basic_string` directly:

Alias	Template	Meaning
`std::string`	`std::basic_string<char>`	C string
`std::wstring`	`std::basic_string<wchar_t>`	Wide character string
`std::u8string`	`std::basic_string<char8_t>`	UTF-8 string
`std::u16string`	`std::basic_string<char16_t>`	UTF-16 string
`std::u32string`	`std::basic_string<char32_t>`	UTF-32 string

There's also a `pmr` version to change how memory is allocated:

Alias	Template	Meaning
`std::pmr::string`	`std::pmr::basic_string<char>`	C string
`std::pmr::wstring`	`std::pmr::basic_string<wchar_t>`	Wide character string

Alias	Template	Meaning
`std::pmr::u8string`	`std::pmr::basic_string<char8_t>`	UTF-8 string
`std::pmr::u16string`	`std::pmr::basic_string<char16_t>`	UTF-16 string
`std::pmr::u32string`	`std::pmr::basic_string<char32_t>`	UTF-32 string

Whichever we choose, the class “owns” the memory that the string is stored in. That means it allocates memory when needed and deallocates it in the destructor. It also provides a bunch of member functions to perform common operations on the string. Here’s a sampling of that functionality:

```
#include <string>

void Foo()
{
    // Allocate memory for the string
    std::string s{ "hello world" };

    // Read and write individual characters
    s[0] = 'H';
    s[6] = 'W';
    DebugLog(s); // Hello World

    // Get a NUL-terminated const pointer to the first
character (a C string)
    const char* cs = s.c_str();
    DebugLog(cs); // Hello World
```

```
    // Get a non-const pointer to the first character
    char* d = s.data();
    DebugLog(d); // Hello World

    // Check if the string is empty
    DebugLog(s.empty()); // false

    // Get the number of characters in the string
    DebugLog(s.size()); // 11
    DebugLog(s.length()); // 11

    // Check how much capacity is there to hold characters
    DebugLog(s.capacity()); // Maybe 15

    // Allocate enough memory to hold a certain number of
characters
    // Note: cannot be used to shrink the string
    s.reserve(128);
    DebugLog(s.capacity()); // At least 128

    // Request reducing allocated memory to just enough to
hold the string
    s.shrink_to_fit();
    DebugLog(s.capacity()); // Maybe 15

    // Add a character to the end
    s.push_back('!');
    DebugLog(s); // Hello World!
```

```
    // Check if the string starts with another string
    DebugLog(s.starts_with("Hello")); // true

    // Replace 1 character starting at index 5 with a comma
and a space
    s.replace(5, 1, ", ");
    DebugLog(s); // Hello, World!

    // Get a string of 5 characters starting at index 7
    std::string ss{ s.substr(7, 5) };
    DebugLog(ss); // World

    // Find an index of a string in the string
    std::string::size_type i = s.find("llo");
    DebugLog(i); // 2

    // Copy the string to another string
    std::string s2{ "other" };
    DebugLog(s2); // other
    s2 = s;
    DebugLog(s2); // Hello, World!

    // Compare strings' characters with overloaded
operators
    DebugLog(s == s2); // true

    // Empty the string
    s.clear();
    DebugLog(s); //
} // Destructor deallocates the string's memory
```

There are also some functions outside of the class that operate on `std::basic_string` objects:

```
#include <string>

// Parse a float out of a string
// Throws an exception upon failure
std::string s{ "3.14" };
float f = std::stof(s);
DebugLog(f); // 3.14

// Convert a double to a string
std::string s2{ std::to_string(3.14) };
DebugLog(s2); // 3.140000

// Check if a string is empty
DebugLog(std::empty(s)); // false

// Get a non-const pointer to the first character
char* d = std::data(s);
DebugLog(d); // 3.14
```

Lastly, there is a user-defined literal in the `std::literals::string_literals` namespace to create strings. The `s` suffix is overloaded to create a string based on the type of characters it's applied to:

```
#include <string>
```

```
using namespace std::literals::string_literals;

// Plain string literals create a std::string
std::string s{ "hello"s };

// char8_t string literals create a UTF-8 string
std::u8string s8{ u8"hello"s };
```

Locale and Codecvt

Next up is `<locale>` to help with localization. The `std::locale` class indentifies a locale like `CultureInfo` does in C#. Its member functions and other functions in `<locale>` allow us to perform operations within the context of that locale:

```
#include <string>
#include <locale>

// Construct a locale for a specific locale name
std::locale loc{ "en_US.UTF-8" };

// Lexicographically compare strings with the overloaded
() operator
std::string a{ "apple" };
std::string b{ "banana" };
DebugLog(loc(a, b)); // true

// Check if a character is in a category for this locale
DebugLog(std::isspace(' ', loc)); // true
DebugLog(std::islower('a', loc)); // true
DebugLog(std::isdigit('1', loc)); // true

// Convert between uppercase and lowercase in this locale
DebugLog(std::toupper('a', loc)); // A
DebugLog(std::tolower('Z', loc)); // z
```

Later in the book we'll look at I/O and see how we can use `std::locale` to localize value categories like time and money.

In the meantime, let's look at `wstring_convert` and `wbuffer_convert` which work with `<codecvt>` to provide conversion facilities between different string formats like UTF-8 and UTF-16. These functions and the `<codecvt>` header were deprecated in C++17 and there will presumably be a replacement at some point in the future. For now, we can use them like this example that converts "😎👍" between UTF-8 and UTF-16:

```
#include <string>
#include <locale>
#include <codecvt>

void Foo()
{
    // Emojis as UTF-8 and UTF-16
    std::string u8 = "\xf0\x9f\x98\x8e\xf0\x9f\x91\x8d";
    std::u16string u16 = u"\xd83d\xde0e\xd83d\xdc4d";

    // Make a converter from UTF-8 to UTF-16

std::wstring_convert<std::codecvt_utf8_utf16<char16_t>,
char16_t> u8u16{};

    // Use it to convert from UTF-8 to UTF-16
    std::u16string toU16 = u8u16.from_bytes(u8);
    DebugLog("Success?", u16 == toU16); // true
    DebugLog("UTF-16 size", toU16.size()); // 4
```

```
    for (uint32_t c : toU16)
    {
        DebugLog(c);
        // Outputs:
        // 55357
        // 56846
        // 55357
        // 56397
    }

    // Make a converter from UTF-16 to UTF-8

std::wstring_convert<std::codecvt_utf8_utf16<char16_t>,
char16_t> u16u8 {};

    // Use it to convert UTF-16 to UTF-8
    std::string toU8 = u16u8.to_bytes(u16);
    DebugLog("Success?", u8 == toU8); // true
    DebugLog("UTF-8 size", toU8.size()); // 8
    for (uint32_t c : toU8)
    {
        DebugLog(c);
        // Outputs:
        // 4294967280
        // 4294967199
        // 4294967192
        // 4294967182
        // 4294967280
```

```
        // 4294967199
        // 4294967185
        // 4294967181
    }
}
```

Format

C++20 adds the `<format>` header to make formatting data as strings easier and safer than existing methods like `sprintf` in the C Standard Library. The `std::format` function is rather similar to string interpolation in C#: `$"Score: {score}"`.

```
#include <string>
#include <format>
#include <locale>

// Format a string
int score = 123;
std::string str{ std::format("Score: {}", score) };
DebugLog(str); // Score: 123

// Format a string for a specific locale
std::locale loc{ "en_US.UTF-8" };
str = std::format(loc, "Score: {}", score);
DebugLog(str); // Score: 123
```

We can specialize the `std::formatter` class template to enable formatting our own types:

```
#include <format>

struct Vector2
{
```

```
    float X;
    float Y;
};

namespace std
{
    template<class TChar>
    struct std::formatter<Vector2, TChar>
    {
        template <typename TContext>
        auto parse(TContext& pc)
        {
            return pc.end();
        }

        template<typename TContext>
        auto format(Vector2 v, TContext& fc)
        {
            return std::format_to(fc.out(), "({}, {})",
v.X, v.Y);
        }
    };
}

Vector2 v{ 1, 2, 3 };
std::string s{ std::format("Vector: {}", v) };
DebugLog(s); // Vector: (1, 2, 3)
```

String View

C++17 introduces `std::basic_string_view` as a class template that provides a read-only "view" into another string. It's an adapter for string literals and other arrays of characters as well as string classes like `std::basic_string`. Unlike `std::basic_string`, it doesn't "own" the memory that holds the characters. That means it doesn't allocate it or deallocate it but instead acts like a pointer to existing memory and a `size_t` to keep track of the length. As with other pointers, it's important to not use the `std::basic_string_view` after the string it points to is deallocated.

Aliases are provided in parallel with `std::basic_string`:

Alias	**Template**	**Meaning**
`std::string_view`	`std::basic_string_view<char>`	View of C string
`std::wstring_view`	`std::basic_string_view<wchar_t>`	View of wide character string
`std::u8string_view`	`std::basic_string_view<char8_t>`	View of UTF-8 string
`std::u16string_view`	`std::basic_string_view<char16_t>`	View of UTF-16 string
`std::u32string_view`	`std::basic_string_view<char32_t>`	View of UTF-32 string

Here's how to use them:

```cpp
#include <string>
#include <string_view>

// A simple array of characters
const char cs[] = "C String";

// A view into the array of characters
std::string_view svcs{ cs };

// A std::basic_string
std::string bs{ "std::string" };

// A view into the std::basic_string
std::string_view svbs{ bs };

// Query the string's size
DebugLog(svcs.empty()); // false
DebugLog(svcs.size()); // 8
DebugLog(svcs.length()); // 8

// Read characters
DebugLog(svcs[2]); // S
DebugLog(svcs[100]); // Throws std::out_of_range exception
DebugLog(svcs.front()); // C
DebugLog(svcs.back()); // g
DebugLog(svcs.data()); // C String

// Copy part of the string
char buf[4] = { '\0' };
```

```
svcs.copy(buf, 3, 2);
DebugLog(buf); // Str

// Get a view of a sub-string. Does not copy characters.
std::string_view sub{ svcs.substr(5, 3) };
DebugLog(sub); // ing

// Compare string views' characters
DebugLog(svcs.compare(svbs)); // -1
DebugLog(svcs == svbs); // false

// C++20: check if the string starts or ends with a sub-
string
DebugLog(svcs.starts_with("C Str")); // true
DebugLog(svcs.ends_with("ING")); // false

// Find a sub-string's index
DebugLog(svcs.find("Str")); // 2

// Reduce the view by moving the view's pointer forward
// Does not modifiy the string
svcs.remove_prefix(2);
DebugLog(svcs); // String

// Reduce the view by reducing the view's size
// Does not modifiy the string
svcs.remove_suffix(3);
DebugLog(svcs); // Str
```

Again paralleling `std::basic_view`, there are also some functions outside of the `std::basic_string_view` class that operate on `std::basic_string_view` objects:

```
#include <string>
#include <string_view>

const char cs[] = "C String";
std::string_view svcs{ cs };

// Check if a string view is empty
DebugLog(std::empty(svcs)); // false

// Get a pointer to the first character
const char* d = std::data(svcs);
DebugLog(d); // C String
```

There is also a user-defined literal in the `std::literals::string_view_literals` namespace to create string views with the `sv` suffix. It's an inline namespace of `std::literals`, so we can avoid a little typing:

```
#include <string>
#include <string_view>

using namespace std::literals;

const char cs[] = "C String";
std::string_view svcs{ cs };
```

```
// Plain string literals create a std::string_view
std::string_view s{ "hello"sv };

// char8_t string literals create a UTF-8 string view
std::u8string_view s8{ u8"hello"sv };
```

Like `std::basic_string`, using `std::basic_string_view` is vastly more convenient than using a C-style array of characters. Since both `std::basic_string` and arrays of characters are implicitly and cheaply converted to `std::basic_string_view`, we can use this type to gain that convenience while supporting different kinds of strings.

The closest C# equivalent to this is `ReadOnlySpan<char>` as it provides a "view" into the characters of a `String`. We'll see C++'s generalized `std::span` equivalent to this later in the book.

Regex

Finally for this chapter we have regular expressions in the `<regex>` header. The `std::basic_regex` class template supports several types of syntax via `std::regex::awk`, `std::regex::grep`, `std::regex::ECMAScript`, and so forth:

```
#include <string>
#include <regex>

// A regular expression for YYYY-MM-DD dates with
ECMAScript grammar
// Each part of the date is captured in a group
std::regex re{
    "(\\d{4})-(\\d{2})-(\\d{2})",
    std::regex_constants::ECMAScript };

// Check if a string matches and get the results of the
match
std::cmatch results{};
DebugLog(std::regex_match("before 2021-03-15 after",
results, re)); // true
DebugLog(results.size()); // 4
DebugLog(results[0]); // 2021-03-15 (sub-string that
matched)
DebugLog(results[1]); // 2021 (first group)
DebugLog(results[2]); // 03 (second group)
DebugLog(results[3]); // 15 (third group)
```

```
// Replace the part of a string that matches
std::basic_string s{
    std::regex_replace(
        std::string{ "before 2021-03-15 after" }, re,
"YYYY-MM-DD") };
DebugLog(s); // before YYYY-MM-DD after
```

A wide variety of overloads are available to support various types of strings, sub-strings, character types, case sensitivity, and so forth. In particular, `std::cmatch` in the above example is an alias to the `std::match_results` class template for C-style strings. Other aliases for wide character strings and `std::basic_string` are available.

The C# equivalent of this are classes like `Regex` and `Match` in the `System.Text.RegularExpressions` namespace.

Conclusion

The C++ Standard Library layers quite a lot of functionality on top of a very humble basis. Simple characters and arrays of characters are extended all the way up to regular expressions, string classes, and string views. In between we have functionality for quick and convenient serialization, parsing, and localization.

As is usual for the Standard Library, all of this is done via the specialization of templates. We choose the most optimal version at compile time rather than relying on runtime strategies like virtual functions. We can specialize any of these templates to support new types of strings or to format our own app's types and reap all the same benefits that standardized types like `std::basic_string` do.

45. Array Containers Library

Vector

Let's start with one of the most commonly-used container types: `std::vector`. This class, found in `<vector>`, is the equivalent of `List` in C# as it implements a dynamic array. Here's a sampling of its API:

```
#include <vector>

void Foo()
{
    // Create an empty vector of int
    std::vector<int> v{};

    // Add an element to the end
    v.push_back(123);

    // Construct an element in place at the end
    v.emplace_back(456);

    // Get size information
    DebugLog(v.empty()); // false
    DebugLog(v.size()); // 2
    DebugLog(v.capacity()); // At least 2
    DebugLog(v.max_size()); // Maybe 4611686018427387903

    // Request changes to capacity
```

```
    v.reserve(100); // Note: can't shrink
    DebugLog(v.capacity()); // 100
    v.shrink_to_fit();
    DebugLog(v.capacity()); // Maybe 2

    // Shrink to just the first element
    v.resize(1);

    // Add two defaulted elements to the end
    v.resize(3);

    // Access elements with overloaded index operator
    v[2] = 789;
    DebugLog(v[0], v[1], v[2]); // 123, 0, 789

    // Access first and last elements
    DebugLog(v.front()); // 123
    v.back() = 1000;
    DebugLog(v[2]); // 1000

    // Get a pointer to the first element
    int* p = v.data();
    DebugLog(p[0], p[1], p[2]); // 123, 0, 1000

    // Create a vector with four elements
    std::vector<int> v2{ 2, 4, 6, 8 };

    // Compare vectors' elements
```

```
    DebugLog(v == v2); // false

    // Replace the elements of v with the elements of v2
    v = v2;
    DebugLog(v.size()); // 4
    DebugLog(v[0], v[1], v[2], v[3]); // 2, 4, 6, 8
} // Destructors free memory of v and v2
```

In C++20, the `<vector>` header also provides a couple of non-member functions to erase elements from a `std::vector`:

```
#include <vector>

std::vector<int> v1{ 100, 200, 200, 200, 300 };

// Erase every element that equals 200
std::vector<int>::size_type numErased = std::erase(v1,
200);
DebugLog(numErased); // 3
DebugLog(v1.size()); // 2
DebugLog(v1[0], v1[1]); // 100, 300

std::vector<int> v2{ 1, 2, 3, 4, 5 };

// Erase all the even numbers
numErased = std::erase_if(v2, [](int x) { return (x % 2)
== 0; });
DebugLog(numErased); // 2
```

```
DebugLog(v2.size()); // 3
DebugLog(v2[0], v2[1], v2[2]); // 1, 3, 5
```

As is the case with std::string, `std::vector` “owns” the memory that elements are stored in. It allocates the memory when needed by the constructor and functions like `push_back`. It deallocates the memory when it needs to grow, in functions like `shrink_to_fit`, and especially in the destructor. Unlike the managed `List` class in C#, it’s not garbage-collected since C++ has no garbage collector. We can simulate that with the reference-counted std::shared_ptr if needed.

Array

Sometimes we want to use an array but don't want the overhead of a dynamic array or we want to allocate it on the stack instead of the heap. In C#, we'd use `stackalloc` or a `fixed` buffer. If heap allocation was acceptable, we could also use a managed array. In C++, we could use an array but that has some notable downsides. It degrades to a pointer, its size isn't easily inspected, and it lacks all the convenient member functions that `std::vector` provides.

To address these issues, the `<array>` header provides the `std::array` class template that holds a statically-sized array. While `std::vector` holds a pointer to the first element of the array, a `std::array` holds the actual array. It's the difference between a `struct Vector { int* P; };` and `struct Array { int E[100]; };`. This difference allows `std::array` to be allocated on the stack. It also requires that the size be specified as a template parameter:

```
#include <array>

// Create an array of 3 int elements
std::array<int, 3> a{ 1, 2, 3 };

// Query its size
DebugLog(a.size()); // 3
DebugLog(a.max_size()); // 3
DebugLog(a.empty()); // false

// Read and write its elements
DebugLog(a[0], a[1], a[2]); // 1, 2, 3
```

```
a[0] = 10;
DebugLog(a.front()); // 10
DebugLog(a.back()); // 3

// Get a pointer to the first element
int* p = a.data();
DebugLog(p[0], p[1], p[2]); // 10, 2, 3

// Create another array of 3 int elements
std::array<int, 3> a2{ 10, 2, 3 };

// Compare elements of arrays
DebugLog(a == a2); // true
```

Note that `std::array` doesn't require that it's allocated on the stack. We can easily allocate one on the heap like any other class:

```
#include <array>

std::array<int, 3>* a = new std::array<int, 3>{ 1, 2, 3
};
DebugLog((*a)[0], (*a)[1], (*a)[2]); // 1, 2, 3
delete a;
```

C++20 adds the non-member `std::to_array` function to copy an array into a `std::array`:

```cpp
#include <array>

// Plain array
int a[3] = { 1, 2 };

// Copy the plain array into a std::array
std::array<int, 3> c{ std::to_array(a) };

// Changing one doesn't change the other
a[0] = 10;
c[1] = 20;
DebugLog(a[0], a[1]); // 10, 2
DebugLog(c[0], c[1]); // 1, 20
```

Valarray

The `<valarray>` header is part of the numbers library but also a container. The same can be said for `std::basic_string`. Like `std::vector`, both are backed by arrays. The major difference is the set of member functions which operate on those arrays. As `std::basic_string` has functions for finding sub-strings, `std::valarray` is geared toward operations on every element of a `std::valarray` object or on pairs of elements in two `std::valarray` objects:

```
#include <valarray>

void Foo()
{
    // Create an array of two ints
    std::valarray<int> va1{ 10, 20 };

    // Create another array of two ints
    std::valarray<int> va2{ 10, 30 };

    // Compare element 0 in each, element 1 in each, and
so on
    // Return a valarray of comparison results
    std::valarray<bool> eq{ va1 == va2 };
    DebugLog(eq[0], eq[1]); // true, false

    // Add elements
    std::valarray<int> sums{ va1 + va2 };
```

```
    DebugLog(sums[0], sums[1]); // 20, 50

    // Access elements
    DebugLog(va1[0]); // 10
    va1[1] = 200;
    DebugLog(va1[0], va1[1]); // 10, 200

    // Shift elements 1 toward the front, filling in with
zeroes
    std::valarray<int> shifted{ va1.shift(1) };
    DebugLog(shifted[0], shifted[1]); // 200, 0

    // Shift elements 1 toward the front, rotating around
to the back
    std::valarray<int> cshifted{ va1.cshift(1) };
    DebugLog(cshifted[0], cshifted[1]); // 200, 10

    // Copy all elements to another valarray
    va1 = va2;
    DebugLog(va1[0], va1[1]); // 10, 30

    // Call a function with each element and assign the
return value to it
    std::valarray<int> plusOne{ va1.apply([](int x) {
return x + 1; }) };
    DebugLog(plusOne[0], plusOne[1]); // 11, 33

    // Take 2^4 and 3^2
```

```
    std::valarray<float> bases{ 2, 3 };
    std::valarray<float> powers{ 4, 2 };
    std::valarray<float> squares{ std::pow(bases, powers)
};
    DebugLog(squares[0], squares[1]); // 16, 9
} // Destructors free memory of all valarrays
```

Like `std::vector`, `std::valarray` "owns" the memory that stores its elements. C# doesn't have an equivalent of this class template.

Some helper classes exist to "slice" more than one element out of a `std::valarray` by passing instances of the class to the overloaded `[]` operator:

```
#include <valarray>

std::valarray<int> va1{ 10, 20, 30, 40, 50, 60, 70 };

// A slice that starts at index 1 plus 2 elements with 0
stride
std::slice s{ 1, 2, 0 };

// Slice the valarray to get a slice_array that refers to
the slice
std::slice_array<int> sa{ va1[s] };

// Copy the slice into a new valarray
std::valarray<int> sliced{ sa };
DebugLog(sliced.size()); // 2
```

```
DebugLog(sliced[0], sliced[1]); // 20, 30

// Slice that starts at index 1 with sizes 2 and 3 and
strides 1 and 2
std::gslice g{ 1, {2, 3}, {1, 2} };

// Slice the valarray to get a gslice_array that refers
to the slice
std::gslice_array ga{ va1[g] };

// Copy the slice into a new valarray
std::valarray<int> gsliced{ ga };
DebugLog(gsliced.size()); // 6
DebugLog(gsliced[0], gsliced[1], gsliced[2]); // 20, 40,
60
DebugLog(gsliced[3], gsliced[4], gsliced[5]); // 30, 50,
70
```

Deque

`std::deque`, pronounced like “deck” and located in `<deque>`, is a doubly-ended queue that owns its elements. Internally, it holds a list of arrays but this is hidden by its API which gives the appearance that it’s one contiguous array similar to a `std::vector`. This means element access involves a second indirection, but it’s fast to add and remove elements from the beginning and end of a `std::deque`. C# has no equivalent to this container type. Here’s how to use it:

```
#include <deque>

void Foo()
{
    // Create a deque of three floats
    std::deque<float> d{ 10, 20, 30 };

    // Query its size
    DebugLog(d.size()); // 3
    DebugLog(d.max_size()); // Maybe 4611686018427387903
    DebugLog(d.empty()); // false

    // Access its elements
    DebugLog(d.front()); // 10
    d[1] = 200;
    DebugLog(d[1]); // 200
    DebugLog(d.back()); // 30

    // Add to and remove from the beginning and the front
```

```
    d.push_front(5);
    d.push_back(35);
    DebugLog(d[0], d[1], d[2], d[3], d[4]); // 5, 10,
200, 30, 35
    d.pop_front();
    d.pop_back();
    DebugLog(d[0], d[1], d[2]); // 10, 200, 30

    // Remove all but the first two elements
    d.resize(2);
    DebugLog(d.size()); // 2
    DebugLog(d[0], d[1]); // 10, 200

    // Compare elements of two deques
    std::deque<float> d2{ 10, 200 };
    DebugLog(d == d2); // true

    // Remove all of a particular element value
    std::deque<float>::size_type numErased =
std::erase(d, 10);
    DebugLog(numErased); // 1
    DebugLog(d[0]); // 200

    // Remove all elements that a function returns true
for
    numErased = std::erase_if(d2, [](float x) { return x
< 100; });
    DebugLog(numErased); // 1
```

```
    DebugLog(d2[0]); // 200
}  // Destructors free memory of all deques
```

Queue

Unlike `std::deque`, the `std::queue` class template in `<queue>` is an adapter to provide a queue API to another collection. The default container type is `std::deque`, but other containers with `back`, `front`, `push_back`, and `push_front` member functions may be used. The `std::queue` contains this collection type and provides member functions that are implemented by calls to the contained collection.

```
#include <queue>
#include <deque>

void Foo()
{
    // Explicitly use std::deque<int> to hold elements of
a std::queue of int
    std::queue<int, std::deque<int>> qd{};

    // Use the default collection type, which is
std::deque
    // It's initially empty
    std::queue<int> q{};

    // Add elements to the back
    q.push(10);
    q.push(20);
    q.emplace(30); // In-place construction

    // Query the size
```

```
    DebugLog(q.size()); // 3
    DebugLog(q.empty()); // false

    // Access only the first and last elements
    DebugLog(q.front()); // 10
    DebugLog(q.back()); // 30

    // Remove elements from the front
    q.pop();
    DebugLog(q.size()); // 2
    DebugLog(q.front()); // 20

    // Copy elements to another queue
    std::queue<int> q2{};
    q2 = q;
    DebugLog(q2.size()); // 2
    DebugLog(q2.front()); // 20
    DebugLog(q2.back()); // 30
} // Destructors free memory of all queues
```

C# has an equivalent `Queue` class. Unlike `std::queue`, it's not an adapter for another collection type. Instead, it implements a particular collection internally.

Stack

The other adapter type, similar to `std::queue`, is `std::stack` in the `<stack>` header. It also defaults to `std::deque` as its collection type. Since stacks only operate on the back, more types of collections can be used. All they need to have are `back`, `push_back`, and `pop_back` member functions:

```
#include <stack>
#include <vector>

void Foo()
{
    // Make a stack backed by a std::vector
    std::stack<int, std::vector<int>> sv{};

    // Make a stack backed by the default std::deque
    std::stack<int> s{};

    // Add elements to the back
    s.push(10);
    s.push(20);
    s.emplace(30); // In-place construction

    // Query the size
    DebugLog(s.size()); // 3
    DebugLog(s.empty()); // false
```

```
    // Access only the last element
    DebugLog(s.top()); // 30

    // Remove elements from the back
    s.pop();
    DebugLog(s.size()); // 2
    DebugLog(s.top()); // 20

    // Copy elements to another stack
    std::stack<int> s2{};
    s2 = s;
    DebugLog(s2.size()); // 2
    DebugLog(s2.top()); // 20
} // Destructors free memory of all stacks
```

Also like `Queue`, C# has a `std::stack` equivalent in `Stack`. It also is not an adapter, but a uniquely-implemented container type.

Conclusion

Arrays are so ubiquitous that the C++ Standard Library provides several container types to wrap and access them. `vector`, `array`, and `valarray` join `basic_string` in holding elements in a contiguous block of memory. The `vector` type provides resizing, `array` provides stack allocation and low overhead, and `valarray` provides element-wise access and advanced slicing. C#'s has a close equivalent to `vector` in `List`, but `stackalloc` only supporting unmanaged types limits it quite a lot compared to `array` and there's simply no analog to `valarray`.

The `deque` type is suprisingly useful when we need to cheaply add to and remove from the front of an array. While not completely contiguous in memory, it is still *mostly* contiguous and may represent an acceptable tradeoff given that insertion and removal operations at the front of the collection are now in `O(1)` instead of `O(N)`. C# lacks this collection type.

Finally, there are `queue` and `stack` as adapter types for any collection providing the necessary member functions. The C# `Queue` and `Stack` classes instead mandate particular collection implementations.

46. Other Containers Library

Unordered Map

The `<unordered_map>` header provides C++'s `Dictionary` equivalent: `std::unordered_map`. As with other containers like std::vector, it "owns" the memory that keys and values are stored in. Here's a sampling of the API:

```
#include <unordered_map>

void Foo()
{
    // Hash map of int keys to float values
    std::unordered_map<int, float> ifum{};

    // Add a key-value pair
    ifum.insert({ 123, 3.14f });

    // Read the value that 123 maps to
    DebugLog(ifum[123]); // 3.14

    // Try to read the value that 456 maps to
    // There's no such key, so insert a default-
initialized value
    DebugLog(ifum[456]); // 0

    // Query size
    DebugLog(ifum.empty()); // false
```

```
    DebugLog(ifum.size()); // 2
    DebugLog(ifum.max_size()); // Maybe
768614336404564650

    // Try to read and throw an exception if the key
isn't found
    DebugLog(ifum.at(123)); // 3.14
    DebugLog(ifum.at(1000)); // throws std::out_of_range
exception

    // insert() does not overwrite
    ifum.insert({ 123, 2.2f }); // does not overwrite
3.14
    DebugLog(ifum[123]); // 3.14

    // insert_or_assign() does overwrite
    ifum.insert_or_assign(123, 2.2f); // overwrites 3.14
    DebugLog(ifum[123]); // 2.2

    // emplace() constructs in-place
    ifum.emplace(456, 1.123f);

    // Remove an element
    ifum.erase(456);
    DebugLog(ifum.size()); // 1
} // ifum's destructor deallocates the memory storing
keys and values
```

A `std::unordered_multimap` is also available for when there are potentially multiple of the same key. C# has no equivalent of this class template, but it can be approximated with a `Dictionary<TKey, List<TValue>>`. Here's how to use it:

```
#include <unordered_map>

void Foo()
{
    // Create an empty multimap that maps int to float
    std::unordered_multimap<int, float> ifumm{};

    // Insert two of the same key with different values
    ifumm.insert({ 123, 3.14f });
    ifumm.insert({ 123, 2.2f });

    // Check how many values are mapped to the 123 key
    DebugLog(ifumm.count(123)); // 2

    // C++20: check if there are any values mapped to the
123 key
    DebugLog(ifumm.contains(123)); // true

    // Find one of the key-value pairs for the 123 key
    const auto& found = ifumm.find(123);
    DebugLog(found->first, found->second); // Maybe 123,
3.14

    // Loop over all the key-value pairs for the 123 key
```

```
    auto range = ifumm.equal_range(123);
    for (auto i = range.first; i != range.second; ++i)
    {
        DebugLog(i->first, i->second); // 123, 3.14 and
123, 2.2
    }

    // Remove all the key-value pairs with a given key
    ifumm.erase(123);
    DebugLog(ifumm.size()); // 0
} // ifumm's destructor deallocates key and value memory
```

C++20 adds the usual `std::erase_if` non-member function found with the array container types to work with `std::unordered_map` and `std::unordered_multimap`:

```
#include <unordered_map>

// Create maps with 2 key-value pairs each
std::unordered_multimap<int, float> umm{ {123, 3.14},
{456, 2.2f} };
std::unordered_map<int, float> um{ {123, 3.14}, {456,
2.2f} };

// Erase all the key-value pairs where the key is less
than 200
auto lessThan200 = [](const auto& pair) {
    const auto& [key, value] = pair;
```

```
    return key < 200;
};
std::erase_if(um, lessThan200);
std::erase_if(umm, lessThan200);

// 123 key has 0 values associated with it
DebugLog(um.count(123)); // 0
DebugLog(umm.count(123)); // 0
```

Map

The `<map>` header provides ordered versions of `std::unordered_map` and `std::unordered_multimap`. They're called, naturally, `std::map` and `std::multimap`. The basics of their APIs are very similar to the unordered counterparts, which helps with generic programming, but there are also some differences.

Let's start with `std::map`, which is typically a red-black tree. The closest C# equivalent to this container is `OrderedDictionary`, but that class doesn't support generic key and value types unlike `std::map` and `Dictionary`. Here's a sampling of the `std::map` functionality:

```
#include <map>

void Foo()
{
    // Create a map with three keys
    std::map<int, float> m{ {456, 2.2f}, {123, 3.14},
{789, 42.42f} };

    // Many functions from other containers are available
    DebugLog(m.size()); // 3
    DebugLog(m.empty()); // false
    m.insert({ 1000, 2000.0f });
    m.erase(123);
    m.emplace(100, 9.99f);

    // Ordering by key is guaranteed
```

```
    for (const auto& item : m)
    {
        DebugLog(item.first, item.second);
        // Prints:
        //   100, 9.99
        //   456, 2.2
        //   789, 42.42
        //   1000, 2000
    }
} // m's destructor deallocates key and value memory
```

Like `std::unordered_multimap`, `std::multimap` also has no C# equivelent. We can approximate it with an `OrderedDictionary` whose values are `List<TValue>` objects but with the same downside of `OrderedDictionary` not supporting generic key and value types. Regardless, `std::multimap` supports mapping multiple of the same key and is also typically a red-black tree. Values are stored in insertion order:

```
#include <map>

void Foo()
{
    // Create a multimap with three keys: two are
duplicated
    std::multimap<int, float> mm{ {456, 42.42f}, {123,
3.14f}, {123, 2.2f} };

    // Many functions from other containers are available
```

```
    DebugLog(mm.size()); // 3
    DebugLog(mm.empty()); // false
    mm.insert({ 1000, 2000.0f });
    mm.erase(456);
    mm.emplace(100, 9.99f);

    // Ordering by key is guaranteed
    for (const auto& item : mm)
    {
        DebugLog(item.first, item.second);
        // Prints:
        //   100, 9.99
        //   123, 3.14
        //   123, 2.2
        //   1000, 2000
    }
} // mm's destructor deallocates key and value memory
```

Unordered Set

The `<unordered_set>` header provides the equivalent of `HashSet` in C#: `std::unordered_set`. It's like a `std::unordered_map` except that there are only keys so the API is simpler:

```
#include <unordered_set>

void Foo()
{
    // Create a set with four values
    std::unordered_set<int> us{ 123, 456, 789, 1000 };

    // Many functions from other containers are available
    DebugLog(us.size()); // 4
    DebugLog(us.empty()); // false
    us.insert(2000);
    us.erase(456);
    us.emplace(100);
    DebugLog(us.count(123)); // 1
} // us's destructor deallocates value memory
```

A `std::unordered_multiset` is also available to support multiple of the same value. There's no C# version of this, but a `Dictionary<TKey, int>` can be used to approximate it by using the value to count the number of keys. That takes additional memory and is somewhat awkward to use compared the straightforward API of `std::unordered_multiset`:

```
#include <unordered_set>

void Foo()
{
    // Create a multiset with six values: two are
duplicated
    std::unordered_multiset<int> ums{ 123, 456, 123, 789,
1000, 1000 };

    // Many functions from other containers are available
    DebugLog(ums.size()); // 6
    DebugLog(ums.empty()); // false
    ums.insert(2000);
    ums.erase(123); // erases both
    ums.emplace(100);
    DebugLog(ums.count(1000)); // 2
} // ums's destructor deallocates value memory
```

Set

As with maps, ordered versions of the set classes are available. Predictably, the `<set>` header provides `std::set` and `std::multiset`. Both are implemented with more red-black trees. C# has a `SortedSet` class equivalent to `std::set` but not equivalent to `std::multiset`.

The APIs of both `std::set` and `std::multiset` are also very similar to their unordered counterparts. Here's `std::set`:

```
#include <set>

void Foo()
{
    // Create a set with four values
    std::set<int> s{ 123, 456, 789, 1000 };

    // Many functions from other containers are available
    DebugLog(s.size()); // 4
    DebugLog(s.empty()); // false
    s.insert(2000);
    s.erase(456);
    s.emplace(100);
    DebugLog(s.count(123)); // 1

    // Ordering by key is guaranteed
    for (int x : s)
    {
```

```
        DebugLog(x);
        // Prints:
        //   100
        //   123
        //   789
        //   1000
        //   2000
    }
} // s's destructor deallocates value memory
```

And here's `std::multiset`:

```
#include <set>

void Foo()
{
    // Create a multiset with six values: two are
duplicated
    std::multiset<int> ms{ 123, 456, 123, 789, 1000, 1000
};

    // Many functions from other containers are available
    DebugLog(ms.size()); // 6
    DebugLog(ms.empty()); // false
    ms.insert(2000);
    ms.erase(123); // erases both
    ms.emplace(100);
    DebugLog(ms.count(1000)); // 2
```

```
    // Ordering is guaranteed
    for (int x : ms)
    {
        DebugLog(x);
        // Prints:
        //   100
        //   456
        //   789
        //   1000
        //   1000
        //   2000
    }
} // ms's destructor deallocates value memory
```

List

Next up is the `<list>` header and it's `std::list` class template that implements a doubly-linked list. This is equivalent to the `LinkedList` class in C#. It's API is similar to `std::vector` except that operations on the front of the list are also supported and indexing isn't allowed since that would require an expensive walk of the list:

```
#include <list>

void Foo()
{
    // Create an empty list
    std::list<int> li{};

    // Add some values
    li.push_back(456);
    li.push_front(123);

    // Grow by inserting default-initialized values
    li.resize(5);

    // Query size
    DebugLog(li.empty()); // false
    DebugLog(li.size()); // 5

    // Indexing isn't supported. Loop instead.
    for (int x : li)
```

```
    {
        DebugLog(x);
        // Prints:
        //   123
        //   456
        //   0
        //   0
        //   0
    }

    // Remove values
    li.pop_back();
    li.pop_front();

    // Special operations
    li.sort();
    li.remove(0); // remove all zeroes
    li.remove_if([](int x) { return x < 200; }); //
remove all under 200
} // li's destructor deallocates value memory
```

Forward List

A singly-linked list, `std::forward_list`, is also provided via `<forward_list>`. C# doesn't provide an equivalent container. The API is like the reverse of `std::vector` since only operations on the *front* of the list are supported. Unlike most other containers, a `size` function isn't provided since it would require walking the entire list to count nodes:

```
#include <forward_list>

void Foo()
{
    // Create an empty list
    std::forward_list<int> li{};

    // Add some values
    li.push_front(123);
    li.push_front(456);

    // Grow by inserting default-initialized values
    li.resize(5);

    // Query size
    DebugLog(li.empty()); // false

    // Indexing isn't supported. Loop instead.
    for (int x : li)
    {
```

```
        DebugLog(x);
        // Prints:
        //   123
        //   456
        //   0
        //   0
        //   0
    }

    // Remove values
    li.pop_front();

    // Special operations
    li.sort();
    li.remove(0); // remove all zeroes
    li.remove_if([](int x) { return x < 200; }); //
remove all under 200
} // li's destructor deallocates value memory
```

Conclusion

The C++ Standard Library provides a robust and consistent offering of non-array collection types to go along with array collection types like `std::vector`. The APIs are all very similar, which is great for generic programming as the collection type can easily be made into a type parameter.

Whether we need a set, map, or list, ordering or hashing, and even support for duplicate keys or values, a class template is on offer. In contrast, C#'s offerings are more limited as there's sometimes no generic version, no support for duplicate keys, or no class that handles ordering. These may be less-common use cases, but it's nice to have standardized tools available when needed.

47. Containers Library Wrapup

Iterators

C# containers like `List` implement the `IEnumerable<T>` interface. This means they provide a `GetEnumerator` method that returns an `IEnumerator<T>`. This in turn provides a `Current` property to access the current element and a `MoveNext` method to move to the next element.

C++ containers provide the same support for this abstract form of traversing a collection, but they call the object keeping track of the traversal an “iterator” instead of an “enumerator.” Regardless of the container type, it’ll have two member type aliases: `iterator` and `const_iterator`. Here’s how we use them the way we’d manually traverse a collection vith `IEnumerator<T>` in C#:

```
#include <vector>

// Create a vector with three elements
std::vector<int> v{ 1, 2, 3 };

// Call begin() to get an iterator that's at the first
element: 1
// Call end() to get an iterator that's one past the last
element: 3
// Use the overloaded pre-increment operator to advance
to the next element
for (std::vector<int>::iterator it{ v.begin() }; it !=
v.end(); ++it)
```

```
{
    // Use the overloaded dereference operator to get the
element
    DebugLog(*it); // 1 then 2 then 3
}
```

It's worth noting that while both the pre-increment and post-increment operators are overloaded for iterator types, the pre-increment operator is always at least as fast as the post-increment operator. They may be equally fast, but it's a good habit to use the pre-increment operator when manually using iterators.

There are a few variations of the above canonical loop that are commonly seen:

```
#include <vector>

std::vector<int> v{ 1, 2, 3 };

// Use the free functions begin() and end() instead of
the member functions
for (std::vector<int>::iterator it{ std::begin(v) }; it
!= std::end(v); ++it)
{
    DebugLog(*it); // 1 then 2 then 3
}

// Use cbegin() and cend() to get constant (i.e. read-
only) iterators
```

```
for (
    std::vector<int>::const_iterator it{ v.cbegin() };
    it != v.cend();
    ++it)
{
    DebugLog(*it); // 1 then 2 then 3
}

// Use auto to avoid the long type name
for (auto it{ v.cbegin() }; it != v.cend(); ++it)
{
    DebugLog(*it); // 1 then 2 then 3
}
```

The design of iterators and functions like `begin` and `end` make all of the container types compatible with range-based for loops. Since they were introduced in C++11 it's now far less common to see these verbose loops that manually control iterators in the simple, and most common, "start to end" fashion. Instead, we just use a `for` loop and let the compiler generate the same code as the manual version:

```
#include <vector>

std::vector<int> v{ 1, 2, 3 };

// Non-const copies of every element
for (int x : v)
{
    DebugLog(x); // 1 then 2 then 3
```

```
}

// Non-const references to every element
for (int& x : v)
{
    DebugLog(x); // 1 then 2 then 3
}

// const copies of every element
for (const int x : v)
{
    DebugLog(x); // 1 then 2 then 3
}

// const references to every element
for (const int& x : v)
{
    DebugLog(x); // 1 then 2 then 3
}
```

Note that `auto` can be used instead of `int` in all of the above examples.

Besides the basics of traversing a collection from start to end, many iterator types support more functionality. For example, the iterators of a `std::vector` support random access via the overloaded `it[N]` operator. We'll go into these in-depth in the next chapter.

Allocators

C++ containers allow customization of how they allocate memory in two ways. First, the classic way is to pass an allocator type as a template argument. This defaults to the `std::allocator` class template we saw in the System Integration Library. We can create our own type of allocator though and use it to improve performance, add safety checks, allocate memory out of special pools such as the file system or VRAM, or anything else we'd like to do:

```
#include <list>

// Custom allocator using malloc() and free() from the C
Standard Library
template <class T>
struct MallocFreeAllocator
{
    // This allocator allocates objects of type T
    using value_type = T;

    // Default constructor
    MallocFreeAllocator() noexcept = default;

    // Converting copy constructor
    template<class U>
    MallocFreeAllocator(const MallocFreeAllocator<U>&)
noexcept
    {
    }
```

```
    // Allocate enough memory for n objects of type T
    T* allocate(const size_t n) const
    {
        DebugLog("allocate");
        return reinterpret_cast<T*>(malloc(n *
sizeof(T)));
    }

    // Deallocate previously-allocated memory
    void deallocate(T* const p, size_t) const noexcept
    {
        DebugLog("deallocate");
        free(p);
    }
};

void Foo()
{
    // Use the custom allocator to allocate the list's
memory
    std::list<int, MallocFreeAllocator<int>> li{ 1, 2, 3
};

    for (int x : li)
    {
        DebugLog(x);
```

```
    }
}
```

What exactly this prints, besides at least one “allocate”, then “1”, “2”, and “3”, then at least one “deallocate” depends on the implementation of the `std::list` type, but it’s likely to look something like this:

```
allocate
allocate
allocate
allocate
allocate
1
2
3
deallocate
deallocate
deallocate
deallocate
deallocate
```

If we’d rather construct the allocator ourselves, perhaps to customize it in some way, rather than letting the container construct it, then we can pass it to the container’s constructor:

```
// Create an allocator
MallocFreeAllocator<int> alloc{};
```

```
// Pass it to the constructor to be used
std::list<int, MallocFreeAllocator<int>> li{ {1, 2, 3},
alloc };
```

The good thing about this kind of customization is that no virtual functions are required so we don't take a performance hit compared to direct function calls. The bad part is that the allocator becomes part of the type of the container. The above variable has type `std::list<int, MallocFreeAllocator<int>>` which is different from the default type `std::list<int, std::allocator<int>>`. This disables certain functionality such as using the overloaded `=` operator to assign all the elements of a list to another list or to mix and match iterator types.

The other form of memory allocation customization is designed to make the opposite trade-off. Available since C++17, it takes the performance hit of virtual function calls rather than create discrete types that include the allocator. We've seen this before with std::pmr::basic_string and it's an option that's also available on the other container types. There are class templates such as `std::pmr::vector` and `std::pmr::list` that mirror their non-`pmr` versions nearly identically:

```
#include <list>
#include <memory_resource>

// A std::pmr::memory_resource that uses the new and
delete operators
struct NewDeleteMemoryResource : public
std::pmr::memory_resource
```

```
{
    // Allocate bytes, not a particular type
    virtual void* do_allocate(
        std::size_t numBytes, std::size_t alignment)
override
    {
        return new uint8_t[numBytes];
    }

    // Deallocate bytes
    virtual void do_deallocate(
        void* p, std::size_t numBytes, std::size_t
alignment) override
    {
        delete[](uint8_t*)p;
    }

    // Check if this resource's allocation and
deallocation are compatible
    // with that of another resource
    virtual bool do_is_equal(
        const std::pmr::memory_resource& other) const
noexcept override
    {
        // Compatible if the same type
        return typeid(other) ==
typeid(NewDeleteMemoryResource);
    }
```

```
};

void Foo()
{
    // Make the memory resource
    NewDeleteMemoryResource mr{};

    // Make the polymorphic allocator backed by the
memory resource
    std::pmr::polymorphic_allocator<int> polyalloc{ &mr
};

    // Pass the polymorphic allocator to the constructor
to be used
    std::pmr::list<int> li1{ {1, 2, 3}, polyalloc };

    for (int x : li1)
    {
        DebugLog(x); // 1 then 2 then 3
    }

    // Class template instantiations are compatible
    // This has a different polymorphic allocator: the
default
    std::pmr::list<int> li2 = li1;

    for (int x : li2)
    {
```

```
        DebugLog(x); // 1 then 2 then 3
    }
}
```

C#'s managed memory model means there's no customization of any container's allocation strategy.

Free Functions

A number of free functions, i.e. those that aren't a member of any class, work on all the container types. We've already seen `std::begin` and `std::end` above to get iterators. We've also seen C++20's `std::erase` and `std::erase_if` for several array and non-array container types.

There's one more free function that works on containers to talk about: `std::swap`. This function swaps the contents of one container with the contents of another container. This can often be performed very efficiently, such as swapping pointers. For example, a simple array type might overload `std::swap` like this:

```
// Forward-declare the array type
template <typename T>
class SimpleArray;

// Forward-declare the std::swap function overload
namespace std
{
    template<class T>
    void swap(SimpleArray<T>& a, SimpleArray<T>& b)
noexcept;
}

// Define the array type
template <typename T>
class SimpleArray
{
```

```
private:
    T* Elements;
    int Length;

    // Allow the std::swap overload to access private
members
    friend void std::swap(SimpleArray<T>& a,
SimpleArray<T>& b) noexcept;

public:

    SimpleArray(int length)
    {
        Elements = new T[length];
        Length = length;
    }

    ~SimpleArray()
    {
        delete[] Elements;
    }

    int GetLength()
    {
        return Length;
    }

    int& operator[](int index)
```

```
    {
        if (index < 0 || index >= Length)
        {
            throw std::out_of_range{"Index out of
bounds"};
        }
        return Elements[index];
    }
};

// Define the std::swap function overload
namespace std
{
    template<class T>
    void swap(SimpleArray<T>& a, SimpleArray<T>& b)
noexcept
    {
        // Swap pointers instead of copying all the
elements
        T* elements = a.Elements;
        a.Elements = b.Elements;
        b.Elements = elements;

        // Swap lengths
        int length = a.Length;
        a.Length = b.Length;
        b.Length = length;
    }
```

```
}

void Foo()
{
    SimpleArray<int> a{ 3 };
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;

    SimpleArray<int> b{ 5 };
    b[0] = 10;
    b[1] = 20;
    b[2] = 30;
    b[3] = 40;
    b[4] = 50;

    for (int i = 0; i < a.GetLength(); ++i)
    {
        DebugLog(a[i]); // 1, 2, 3
    }
    for (int i = 0; i < b.GetLength(); ++i)
    {
        DebugLog(b[i]); // 10, 20, 30, 40, 50
    }

    // Swap the contents of a and b
    std::swap(a, b);
```

```
    for (int i = 0; i < a.GetLength(); ++i)
    {
        DebugLog(a[i]); // 10, 20, 30, 40, 50
    }
    for (int i = 0; i < b.GetLength(); ++i)
    {
        DebugLog(b[i]); // 1, 2, 3
    }
}
```

Overloads like the one we created above for `SimpleArray` exist for all the C++ Standard Library containers. It's even typical to create overloads like the above for any custom container types we create. Usage with types like `std::unordered_set` looks just the same:

```
#include <unordered_set>

// Create collections
std::unordered_set<int> a{ 1, 2, 3 };
std::unordered_set<int> b{ 10, 20, 30, 40, 50 };

// Print initial contents
for (int x : a)
{
    DebugLog(x); // 1, 2, 3
}
for (int x : b)
{
```

```
    DebugLog(x); // 10, 20, 30, 40, 50
}

// Swap contents
std::swap(a, b);

// Print swapped contents
for (int x : a)
{
    DebugLog(x); // 10, 20, 30, 40, 50
}
for (int x : b)
{
    DebugLog(x); // 1, 2, 3
}
```

Exceptions

C++ supports a wide variety of error-handling techniques ranging from simple return codes to exceptions, std::optional, and even the C Standard Library's errno. Most of the C++ Standard Library uses exceptions though. This includes all the container types, such as detecting out-of-bounds conditions like `SimpleArray` did above:

```
#include <vector>

std::vector<int> v{ 1, 2, 3 };

try
{
    int x = v[1000];
    DebugLog(x); // Does not get printed
}
catch (const std::out_of_range& ex)
{
    DebugLog(ex.what()); // Maybe "vector subscript out
of range"
}
```

The same is true for myriad other erroneous operations. For example, calling `front` on an empty `std::vector` throws an exception because there's no way it could possibly return a valid `T&` or a `T&` indicating an error that would work with all types of `T`:

```
#include <vector>

std::vector<int> v{};

try
{
    int x = v.front();
    DebugLog(x); // Does not get printed
}
catch (const std::out_of_range& ex)
{
    DebugLog(ex.what()); // Maybe "front() called on
empty vector"
}
```

This behavior is very similar to C# where exceptions are used as the preferred error-handling mechanism. Still, some C++ codebases and style guides, especially in video games, forbid using exceptions. C++ compilers typically provide a way to disable this language feature and even to change the error-handling mechanism of the C++ Standard Library so that it doesn't use exceptions. Some Standard Library variants, notably EASTL by Electronic Arts, also support enabling and disabling exceptions.

In-Place Construction

We've seen the use of functions with "emplace" in their names, but not yet delved into what that means. So far they've behaved just like other functions such as those with "push" in their names. Now we'll see how they're different:

```
#include <vector>

// A class that logs its lifecycle
struct Noisy
{
    int Val;

    Noisy(int val)
    {
        Val = val;
        DebugLog(Val, "val ctor");
    }

    Noisy(const Noisy& other)
    {
        Val = other.Val;
        DebugLog(Val, "copy ctor");
    }

    ~Noisy()
    {
```

```
        DebugLog(Val, "dtor");
    }
};

std::vector<Noisy> v{};
v.reserve(2);

v.push_back(123);
// Prints:
//   123, val ctor
//   123, copy ctor
//   123, dtor

v.emplace_back(456);
// Prints:
//   456, val ctor
```

Both `push_back` and `emplace_back` add an element to the end of a `std::vector`, but they do it in different ways. The “push” version takes a reference to an element, a `Noisy&` or `Noisy&&` in this case, and copies it to the array that `std::vector` contains. That means this humble function call actually performs several steps.

First, implicitly convert `123`, which is an rvalue because it doesn’t have a name, to a `Noisy&&` rvalue reference by inserting a call to the `Noisy(int)` constructor. This is why we see the “123, val ctor” log message. The call now looks like this:

```
v.push_back(Noisy{123});
```

Second, the implementation of `push_back` copies the parameter to its array of `Noisy` objects. It's as though `push_back` included a line that used "placement `new`" to call the copy constructor with `this` being the appropriate location in its array. This is why we see the "123, copy ctor" log message. Here's a pseudo-code version of how that might look:

```
void push_back(T&& value)
{
    new (&Elements[EndIndex]) Noisy{ value };
}
```

Finally, the temporary `Noisy` created by the implicit conversion in the first step is destroyed at the end of the statement. As a result, we see the "123, dtor" log message.

On the other hand, `emplace_back` does not take the element to add to the `std::vector`. Instead, it takes the arguments to pass to the "placement `new`" operator. It's as though `push_back` is implemented like this:

```
void push_back(int val)
{
    new (&Elements[EndIndex]) Noisy{ val };
}
```

This means there's no implicit conversion and therefore only one object is created. We see this with the "456, val ctor" log message. Of course the real implementation of `push_back` is more complicated so it can work on the arbitrary numbers of parameters required by various element types `T`.

There’s no equivalent to this in C# as all element additions use the “push” style of copying. In the case of managed types, a managed reference is copied. The size of the copy is just a pointer, but another reference to the object is created and must be tracked for future garbage collection. Unmanaged types such as primitives and structs are simply copied and the cost depends on their size. Copying large structs may be expensive.

Span

Similar to `std::basic_string_view`, C++20 introduces a new "view" type that works with any container: `std::span`. This is very similar to the `Span` and `ReadOnlySpan` types in C#. A `ReadOnlySpan` type is unnecessary in C++ as `const` can be used to disable mutating member functions and overloaded operators. Like `std::basic_string_view`, a `std::span` also doesn't "own" the underlying memory that holds the contained values but rather references it like a pointer would. Here's the basic usage:

```
#include <vector>
#include <span>

// Create a container
std::vector<int> v{ 1, 2, 3 };

// Create a span to view the container
std::span<int> s{ v };

// Use the span to get the contents of the container
for (int x : s)
{
    DebugLog(x); // 1 then 2 then 3
}
```

One benefit of `std::span` is that it abstracts the underlying collection type in a similar way to using `IEnumerable<T>` in C#. It uses templates to do this at compile-time rather than runtime and thus

avoids the overhead of interfaces' virtual functions. Its non-`explicit` constructor combined with C++'s implicit conversion allows for any type of container to be passed to a function taking a `std::span`:

```
#include <vector>
#include <span>

void Print(std::span<int> s)
{
    for (int x : s)
    {
        DebugLog(x);
    }
}

std::vector<int> v{ 1, 2, 3 };
Print(v); // 1 then 2 then 3

int a[] = { 1, 2, 3 };
Print(a); // 1 then 2 then 3
```

`std::span` is able to behave like a container because it contains all the usual member functions: `begin`, `end`, `size`, `front`, `back`, etc. We can call them directly or indirectly via features like the range-based `for` loop above. There's also an additional function to get a `std::span` that views just a portion of an existing `std::span`:

```
#include <vector>
#include <span>
```

```cpp
std::vector<int> v{ 1, 2, 3, 4, 5 };

// A span covering the whole container
std::span<int> s{ v };
DebugLog(s.size()); // 5
for (int x : s)
{
    DebugLog(x); // 1, 2, 3, 4, 5
}

// A sub-span of the middle 3 elements
std::span<int> ss{ s.subspan(1, 3) };
DebugLog(ss.size()); // 3
for (int x : ss)
{
    DebugLog(x); // 2, 3, 4
}
```

Conclusion

The containers library is designed in such a way that containers are very compatible with each other and with the C++ language itself. They all have iterators that can be used directly or via range-based `for` loops just like C# container types can be used with `foreach` loops. They consistently handle errors with exceptions, just like C# containers do, but often allow disabling those exceptions. The `std::span` type provides an abstract view of any container just like `Span` does. All in all, there's a lot of similarity between C++ and C#.

Some divergence appears when we start to look at advanced operations. Swapping the contents of containers is implemented very efficiently in C++ via template specialization, which C# lacks. Allocation customization is available in two forms, but unavailable in C#. In-place construction of elements is only possible in C++ due to the ability to "forward" constructor parameters and perform "placement `new`" construction.

Finally, here's a comparison table between each of the container types in the two languages:

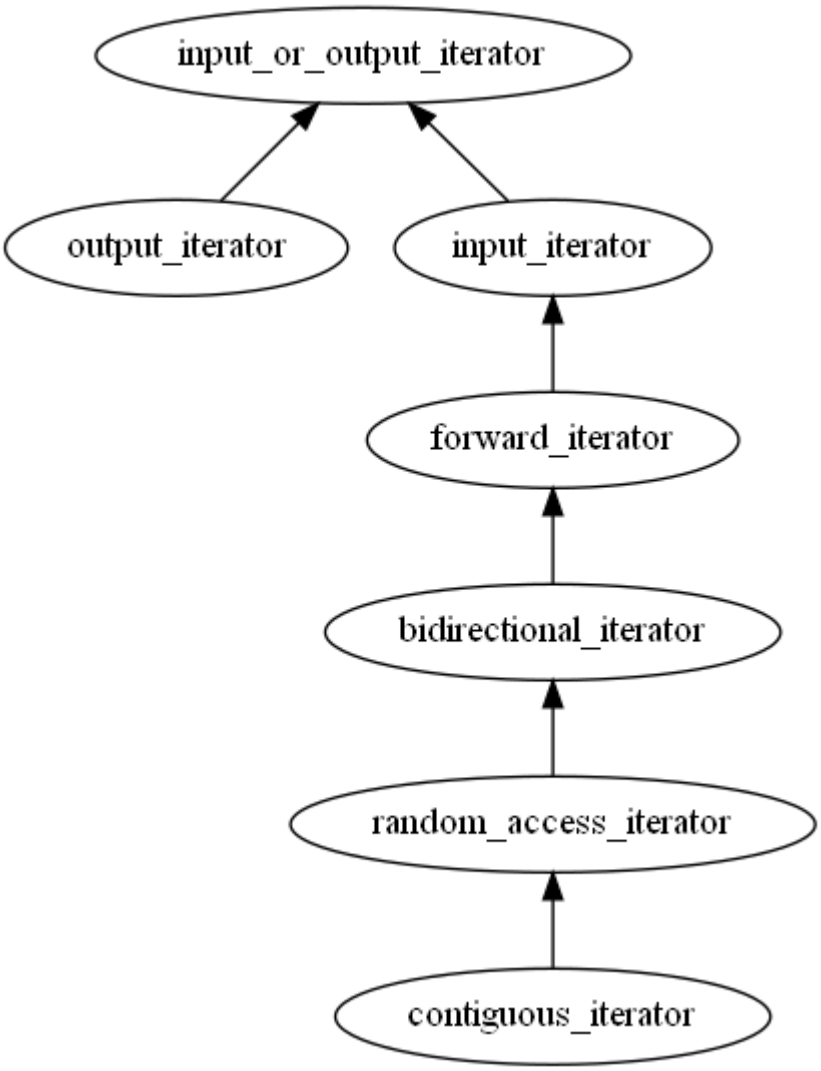
Container	**C++**	**C#**
Dynamic Array	`vector`	`List`
Static Array	`array`	N/A
Bit Array	`bitset`	`BitArray`
Value Array	`valarray`	N/A
Double-Ended Queue	`deque`	N/A
Queue	`queue`	`Queue`

Container	**C++**	**C#**
Stack	`stack`	`Stack`
Hash Map	`unordered_map`	`Dictionary`
Hash Multimap	`unordered_multimap`	N/A
Map	`map`	`OrderedDictionary`
Multimap	`multimap`	N/A
Hash Set	`unordered_set`	`HashSet`
Hash Multiset	`unordered_multiset`	N/A
Set	`set`	`SortedSet`
Multiset	`multiset`	N/A
Doubly-Linked List	`list`	`LinkedList`
Singly-Linked List	`forward_list`	N/A

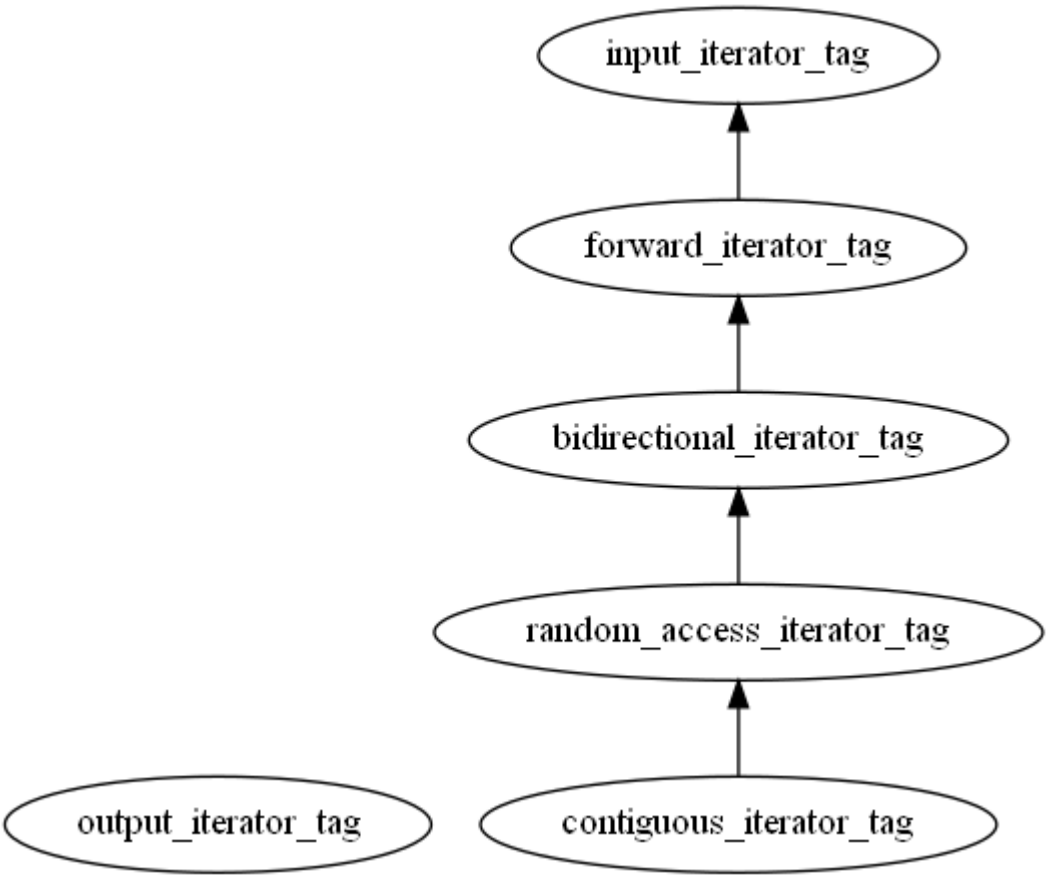
48. Algorithms Library

Iterator

We've seen that containers each have an `iterator` class member that's used like `IEnumerator` in C#: to keep track of iterating over the collection. The `<iterator>` header is the support library for these types. Until C++17 deprecated it, there was a `std::iterator` base class of all the containers' iterator classes. This is now unnecessary with language features like concepts. Specifically, the `<iterator>` header provides a *lot* of concepts to define different kinds of iterators. Here's how they relate to each other:



“Tag” classes are available for most of these concepts. These are empty structs that iterators and other tag classes derive from to form an inheritance hierarchy that matches the concepts:



There are also concepts such as `std::sortable`, `std::mergeable`, and `std::permutable` that are used as requirements for various algorithms we'll see shortly.

C# doesn't have concepts and its `IEnumerator` interface doesn't have "tag" interfaces, so there's really no equivalent of this.

Besides concepts, `<iterator>` provides a lot of iterator-related utilities. This includes a lot of "adapter" classes that change the behavior of iterators. For example, `std::reverse_iterator` has an overloaded `++` operator that moves *backward* instead of forward:

```
#include <iterator>
#include <vector>
```

```cpp
std::vector<int> v{ 1, 2, 3 };

// Adapt iterators to go backward instead of forward when using ++
std::reverse_iterator<std::vector<int>::iterator> it{ v.end() };
std::reverse_iterator<std::vector<int>::iterator> end{ v.begin() };
while (it != end)
{
    DebugLog(*it); // 3 then 2 then 1
    ++it;
}

// Less verbose version using class template argument deduction
for (std::reverse_iterator it{ v.end() };
    it != std::reverse_iterator{ v.begin() };
    ++it)
{
    DebugLog(*it); // 3 then 2 then 1
}

// Even less verbose version using rebgin() and rend()
for (auto it{ v.rbegin() }; it != v.rend(); ++it)
{
    DebugLog(*it); // 3 then 2 then 1
}
```

There's also a `std::back_insert_iterator` that overloades the assignment (`=`) operator to call `push_back` on a collection:

```
#include <iterator>
#include <vector>

// Empty collection
std::vector<int> v{};

// Create an iterator to insert into the std::vector
std::back_insert_iterator<std::vector<int>> it{ v };

// Insert three elements
it = 1;
it = 2;
it = 3;

for (int x : v)
{
    DebugLog(x); // 1 then 2 then 3
}
```

Some functions are also available for common operations on iterators:

```
#include <iterator>
#include <vector>
```

```
std::vector<int> v{ 10, 20, 30, 40, 50 };

// Get the distance (how many iterations) between two
iterators
DebugLog(std::distance(v.begin(), v.end())); // 5

// Advance an iterator by a certain number of iterations
std::vector<int>::iterator it{ v.begin() };
std::advance(it, 2);
DebugLog(*it); // 30

// Get the next iterator
DebugLog(*std::next(it)); // 40
DebugLog(*std::prev(it)); // 20
```

There are also some utility functions for containers:

```
#include <iterator>
#include <vector>

std::vector<int> v{ 10, 20, 30, 40, 50 };

// Check if a container is empty
DebugLog(std::empty(v)); // false

// Get a pointer to a container's data
int* d{ std::data(v) };
DebugLog(*d); // 10
```

A bunch of overloaded operators are provided outside of any particular iterator class to perform binary operations on iterators:

```
#include <iterator>
#include <vector>

std::vector<int> v{ 10, 20, 30, 40, 50 };
std::vector<int>::iterator itA{ v.begin() };
std::vector<int>::iterator itB{ v.end() };

// Subtraction is a synonym for std::distance
DebugLog(itB - itA); // 5

// Inequality and inequality operators compare iteration position
DebugLog(itA == itB); // false
DebugLog(itA < itB); // true
```

Some of this iterator-agnostic functionality is provided by the `System.Linq.Enumerable` class in C#. For example, `Skip` is very similar to `advance` in that it advances an `IEnumerator` forward by a given number of iterations.

Algorithm

The `<algorithm>` header provides a ton of functions that operate on iterators, just as LINQ in C# provides generic algorithms that operate on `IEnumerable` and `IEnumerator`. For example, `std::all_of` is the equivalent of `Enumerable.All`:

```
#include <algorithm>
#include <vector>

std::vector<int> v{ 10, 20, 30, 40, 50 };

bool allEven = std::all_of(
    v.begin(), // Iterator to start at
    v.end(), // Iterator to stop at
    [](int x) { return (x % 2) == 0; }); // Predicate to
call with elements
DebugLog(allEven); // true
```

Right away we see a difference from LINQ: two iterators are passed in instead of an `IEnumerable` with its single `GetEnumerator` method. This makes it easy to operate on a subset of the container such as the middle three elements of an array:

```
#include <algorithm>
#include <vector>

int v[]{ 10, 20, 30, 40, 50 };
```

```
bool allSmall = std::all_of(
    std::begin(v) + 1,
    std::begin(v) + 4,
    [](int x) { return x >= 20 && x <= 40; });
DebugLog(allSmall); // true
```

The same can be done in C#, but only by allocating new managed class objects that implement the `IEnumerable` interface:

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        int[] a = { 10, 20, 30, 40, 50 };

        // Skip allocates a class instance
        IEnumerable<int> skipped = a.Skip(1);

        // Take allocates a class instance
        IEnumerable<int> taken = skipped.Take(3);

        // Operates on only the middle three elements
        bool allSmall = taken.All(x => x >= 20 && x <=
```

```
40);
        Console.WriteLine(allSmall); // true
    }
}
```

LINQ is often avoided, such as in Unity games, due to the high amount of managed allocation and eventual garbage collection involved in creating so many `IEnumerable` class objects. The function calls to these, and the `IEnumerator` objects they return, are all virtual functions which run slower than non-virtual function calls. Additional GC pressure may even be generated by "boxing" of `IEnumerator` structs into managed objects.

C++ algorithms, in contrast, only create lightweight iterators that aren't garbage-collected, are never boxed, and don't use virtual functions. The result is that compilers often totally wipe out the iterators and algorithms resulting in a "raw" loop. That "raw" loop may even be optimized away if the contents of the collection are known at compile time. For example, a command line program version of the above returns `allSmall` as the exit code:

```
#include <algorithm>
#include <iterator>

int main()
{
    int v[]{ 10, 20, 30, 40, 50 };

    bool allSmall = std::all_of(
        std::begin(v) + 1,
        std::begin(v) + 4,
```

```
        [](int x) { return x >= 20 && x <= 40; });
    return allSmall;
}
```

GCC 10.3 with basic optimization (`-O1`) enabled compiles this to the constant `1` (for `true`) on x86-64:

```
main:
        mov     eax, 1
        ret
```

As with LINQ, C++ provides dozens of algorithm functions for many common operations. Here's a sampling of the non-modifying algorithms:

```
#include <algorithm>
#include <vector>

std::vector<int> v1{ 10, 20, 30, 40, 50 };
std::vector<int> v2{ 10, 20, 35, 45, 55 };

auto isEven = [](int x) { return (x % 2) == 0; };

// Query the contents of a vector
DebugLog(std::any_of(v1.begin(), v1.begin(), isEven)); //
true
DebugLog(std::none_of(v1.begin(), v1.begin(), isEven));
// false
```

```
// Get a pair of iterators where the vectors diverge
auto mm{ std::mismatch(v1.begin(), v1.end(), v2.begin())
};
DebugLog(*mm.first, *mm.second); // 30, 35

// Get an iterator to the first matching element
auto firstEven{ std::find_if(v2.begin(), v2.end(),
isEven) };
DebugLog(*firstEven); // 10

// Get an iterator to the first matching element that
doesn't match
auto firstOdd{ std::find_if_not(v2.begin(), v2.end(),
isEven) };
DebugLog(*firstOdd); // 35

// Get an iterator to the first element of an element
sequence
auto seq{ std::search(v1.begin(), v1.end(), v2.begin(),
v2.begin() + 1) };
DebugLog(*seq); // 10
```

Here are some modifying algorithms:

```
#include <algorithm>
#include <vector>
#include <random>
```

```
std::vector<int> v{ 10, 20, 35, 45, 55 };

auto isEven = [](int x) { return (x % 2) == 0; };
auto print = [&]() {
    for (int x : v)
    {
        DebugLog(x);
    }
};

// Remove matching elements by shifting toward the front
// Returns an iterator just after the new end
auto end{ std::remove_if(v.begin(), v.end(), isEven) };
DebugLog(*end); // 45
print(); // 35, 45, 55

// Replace matching elements with a new value
std::replace_if(v.begin(), v.end(), [](int x) { return x
< 50; }, 50);
print(); // 50, 50, 55, 50, 55

// Rotate left by two elements
std::rotate(v.begin(), v.begin() + 2, v.end());
print(); // 55, 50, 55, 50, 50

// Randomly shuffle elements
// Note: random_shuffle() isn't thread-safe and is
```

```
deprecated since C++17
std::random_device rd{};
std::mt19937 gen{ rd() };
std::shuffle(v.begin(), v.end(), gen);
print(); // Some permutation of 55, 50, 55, 50, 50

// Assign a value to every element
std::fill(v.begin(), v.end(), 10);
print(); // 10, 10, 10, 10, 10
```

There are also algorithms related to sorting sequences:

```
#include <algorithm>
#include <vector>

std::vector<int> v{ 35, 45, 10, 20, 55 };

auto isEven = [](int x) { return (x % 2) == 0; };
auto print = [](auto& c) {
    for (int x : c)
    {
        DebugLog(x);
    }
};

// Check if a sequence is sorted
DebugLog(std::is_sorted(v.begin(), v.end())); // false
```

```
// Sort elements until an iterator is reached
std::partial_sort(v.begin(), v.begin() + 2, v.end());
print(v); // 10, 20, 45, 35, 55
DebugLog(std::is_sorted(v.begin(), v.end())); // false

// Sort the whole sequence
std::sort(v.begin(), v.end());
print(v); // 10, 20, 35, 45, 55
DebugLog(std::is_sorted(v.begin(), v.end())); // true

// Binary search a sorted sequence
DebugLog(std::binary_search(v.begin(), v.end(), 45)); //
true

// Merge two sorted sequences into a sorted sequence
std::vector<int> v2{ 15, 25, 40, 50 };
std::vector<int> v3{};
std::merge(
    v.begin(), v.end(), // First sequence
    v2.begin(), v2.end(), // Second sequence
    std::back_insert_iterator<std::vector<int>>{ v3 });
// Output iterator
print(v3); // 10, 15, 20, 25, 35, 40, 45, 55

// Check if a sorted sequence includes another sorted
sequence
// Inclusion doesn't need to be contiguous
bool inc{ std::includes(v3.begin(), v3.end(), v2.begin(),
```

```
v2.end()) };
DebugLog(inc); // true
```

And finally there are some query operations:

```
#include <algorithm>
#include <vector>

std::vector<int> v1{ 35, 45, 10, 20, 55 };
std::vector<int> v2{ 35, 45, 10, 15, 30 };

// Check if two sequences' elements are equal
DebugLog(std::equal(v1.begin(), v1.end(), v2.begin(),
v2.end())); // false
DebugLog(
    std::equal(
        v1.begin(), v1.begin() + 3,
        v2.begin(), v2.begin() + 3)); // true

// Find the min, max, and both of elements in a sequence
DebugLog(*std::min_element(v1.begin(), v1.end())); // 10
DebugLog(*std::max_element(v1.begin(), v1.end())); // 55
auto [minIt, maxIt] = std::minmax_element(v1.begin(),
v1.end());
DebugLog(*minIt, *maxIt); // 10, 55

// Single value versions don't operate on sequences
int a = 10;
```

```
int b = 20;
DebugLog(std::min(a, b)); // 10
DebugLog(std::max(a, b)); // 20
auto [minVal, maxVal] = std::minmax(a, b);
DebugLog(minVal, maxVal); // 10, 20

// Other single value functions
DebugLog(std::clamp(1000, 0, 100)); // 100
std::swap(a, b);
DebugLog(a, b); // 20, 10
```

All of these examples have used `std::vector<int>`, but it's important to know that these algorithms apply to any container type with any element type as long as the requirements of the algorithm are satisfied. This includes container and element types implemented in the C++ Standard Library as well as container and element types we create in our own code:

```
#include <algorithm>

// A custom enum and a function to get enumerator string
names
enum class Element { Earth, Water, Wind, Fire };
const char* GetName(Element e)
{
    switch (e)
    {
        case Element::Earth: return "Earth";
        case Element::Water: return "Water";
```

```
        case Element::Wind: return "Wind";
        case Element::Fire: return "Fire";
        default: return "";
    }
}

// A custom struct
struct PrimalElement
{
    Element Element;
    int Power;
};

// Forward-declare a class that holds an array of the
custom struct
class PrimalElementsArray;

// An iterator type for the custom struct
class PrimalElementIterator
{
    // Keep track of the current iteration position
    PrimalElement* Array;
    int Index;

public:

    PrimalElementIterator(PrimalElement* array, int
index)
```

```
        : Array(array)
        , Index(index)
    {
    }

    // Advance the iterator
    PrimalElementIterator& operator++()
    {
        Index++;
        return *this;
    }

    // Compare with another iterator
    bool operator==(const PrimalElementIterator& other)
    {
        return Array == other.Array && Index ==
other.Index;
    }

    // Dereference to get the current element
    PrimalElement& operator*()
    {
        return Array[Index];
    }
};

// A class that holds an array of the custom struct
class PrimalElementsArray
```

```
{
    PrimalElement Elements[4];

public:

    PrimalElementsArray()
    {
        Elements[0] = PrimalElement{ Element::Earth, 50
};
        Elements[1] = PrimalElement{ Element::Water, 20
};
        Elements[2] = PrimalElement{ Element::Wind, 10 };
        Elements[3] = PrimalElement{ Element::Fire, 75 };
    }

    // Get an iterator to the first element
    PrimalElementIterator begin()
    {
        return PrimalElementIterator{ Elements, 0 };
    }

    // Get an iterator to one past the last element
    PrimalElementIterator end()
    {
        return PrimalElementIterator{ Elements, 4 };
    }
};
```

```
// Create our custom array type
PrimalElementsArray pea{};

// Use std::find_if to find the PrimalElement with more
than 50 power
PrimalElementIterator found{
    std::find_if(
        pea.begin(),
        pea.end(),
        [](const PrimalElement& pe) { return pe.Power >
50; }) };
DebugLog(GetName((*found).Element), (*found).Power); //
Fire, 75
```

Numeric

Finally for this chapter, we'll revisit the numbers library by looking at `<numeric>`. It turns out that it has some number-specific generic algorithms. Here's a few of them:

```
#include <numeric>
#include <vector>

std::vector<int> v{};
v.resize(5);

auto print = [](auto& c) { for (int x : c) DebugLog(x);
};

// Initialize with sequential values starting at 10
std::iota(v.begin(), v.end(), 10);
print(v); // 10, 11, 12, 13, 14

// Sum the range starting at 100
DebugLog(std::accumulate(v.begin(), v.end(), 100)); //
160

// Sum in an arbitrary order
DebugLog(std::reduce(v.begin(), v.end(), 100)); // 160

// C++17: transform pairs of elements and then reduce in
an arbitrary order
```

```
// Equivalent to 1000000 + 10*10 + 11*10 + 12*10 + 13*10
+ 14*10
DebugLog(
    std::transform_reduce(
        v.begin(), v.end(), // Sequence
        1000000, // Initial value
        [](int a, int b) { return a + b; }, // Reduce
function
        [](int x) { return x*10; }) // Transform function
); // 1000600

// Output sums up to current iteration
std::vector<int> sums{};
std::partial_sum(
    v.begin(), v.begin() + 3, // Sequence
    std::back_insert_iterator{ sums }); // Output
iterator
print(sums); // 10, 21, 33
```

Conclusion

Both languages have a wide variety of generic algorithms but they differ quite a bit in implementation. That ranges from the trivial naming differences of enumerators and iterators to the giant performance gulf between LINQ and C++ algorithm functions in `<algorithm>` and `<numeric>`.

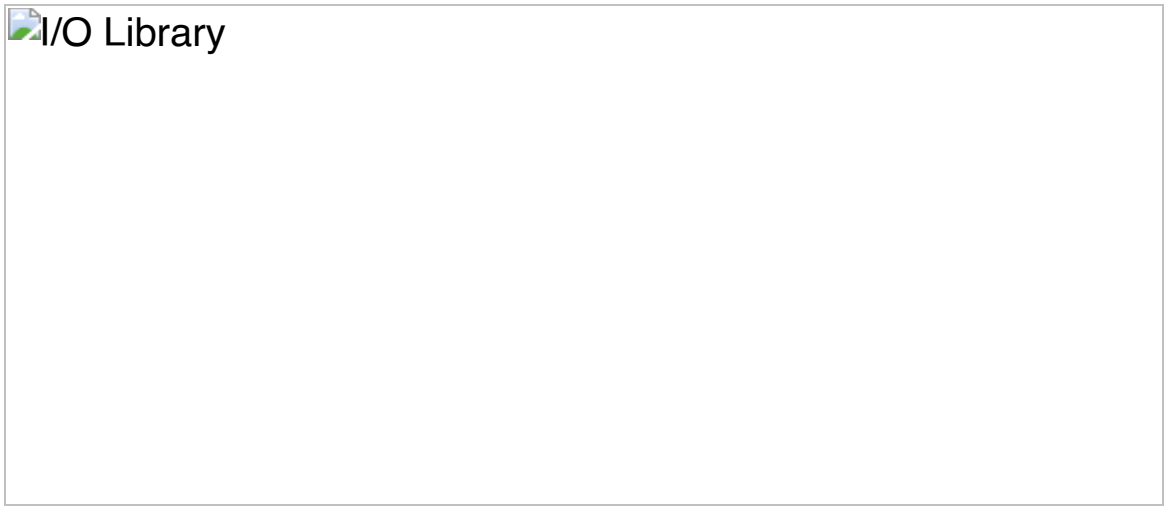
It's hard to overstate just how many generic algorithms are available in the C++ Standard Library. This is especially true when looking at the huge number of permutations of each of these functions. It's common to see five or even ten overloads of these to customize for a wide variety of parameters running the gamut from simple versions to extremely generic and flexible versions. That's another difference with C# where LINQ functions typically have just one or two overloads.

The design of the language, especially the very powerful support for compile-time generic programming via templates, combines with the iterator paradigm to enable all of this functionality on all of the many container types but also all of the container types we might implement in our own code to suit our own needs. We inherit the same high level of optimization that C++ Standard Library types receive, which gives us little excuse for writing a lot of "raw" loops.

49. Ranges and Parallel Algorithms' href

Library Layout and iosfwd

The I/O library subset of the broader C++ Standard Library contains several header files that often `#include` each other. Here's how those relationships look:



The most basic usage of the I/O library is to `#include <iosfwd>`. This header provides "forward" declarations of I/O types. These can then be named, such as by pointer or reference types. They can't be used by value or by accessing any of their members. `<iosfwd>` really just exists to speed up compilation when I/O types only need to be named and their full definitions, which are slower to compile, aren't needed.

We'll see all the types in `<iosfwd>` as we go through each of the headers that `#include` it and then provide definitions.

ios

The `<ios>` header contains basic tools that the rest of the I/O library uses. First up, there's `std::ios_base` which serves as the abstract base class of all the I/O "stream" classes. We'll see what those classes look like soon, but suffice to say a "stream" is an abstract input and/or output that can be backed by a file, "standard output", a string, and so forth. It's very similar to the C# `Stream` abstract base class.

Here's some of what `std::ios_base` provides:

```
#include <ios>
#include <locale>

void Goo(std::ios_base& base)
{
    // Get the flags that control formatting
    std::ios_base::fmtflags f{ base.flags() };
    DebugLog((f & std::ios_base::dec) != 0); // Maybe
true
    DebugLog((f & std::ios_base::hex) != 0); // Maybe
false
    DebugLog((f & std::ios_base::boolalpha) != 0); //
Maybe false

    // Set and unset a format flag
    base.setf(std::ios_base::boolalpha);
    DebugLog((base.flags() & std::ios_base::boolalpha) !=
```

```
0); // true
    base.unsetf(std::ios_base::boolalpha);
    DebugLog((base.flags() & std::ios_base::boolalpha) !=
0); // false

    // Set and get floating-point precision
    base.precision(2);
    DebugLog(base.precision()); // 2

    // Set and get the minimum number of characters that
some operations print
    base.width(10);
    DebugLog(base.width()); // 10

    // Set and get the locale
    base.imbue(std::locale{ "de-DE" });
    DebugLog(base.getloc().name()); // de-DE

    try
    {
        throw std::ios_base::failure{ "some I/O error" };
    }
    catch (const std::ios_base::failure& ex)
    {
        DebugLog(ex.what()); // some I/O error
    }

    // Ways of opening streams
```

```
    // These are bit flags to form a mask
    std::ios_base::openmode mode{
        std::ios_base::app | // Append
        std::ios_base::binary | // Binary
        std::ios_base::in | // Read
        std::ios_base::out | // Write
        std::ios_base::trunc | // Overwrite
        std::ios_base::ate // Open at end of stream
    };

    // Bit flags forming the state of a stream
    std::ios_base::iostate state{
        std::ios_base::goodbit | // No error
        std::ios_base::badbit | // Unrecoverable error
        std::ios_base::failbit | // Operation failed
(e.g. formatting failed)
        std::ios_base::eofbit // End of stream
    };

    // Directions to seek
    std::ios_base::seekdir dir = std::ios_base::beg; //
Beginning of stream
    dir = std::ios_base::end; // End of stream
    dir = std::ios_base::cur; // From the current
position
}
```

There's also `std::char_traits`, which is a class template with static functions that provide functionality for operations on particular kinds of characters:

```
#include <ios>

// Single-character operations
DebugLog(std::char_traits<char>::eq('a', 'a')); // true
DebugLog(std::char_traits<char>::eof()); // -1

// Copy multiple characters
char buf[5];
std::char_traits<char>::copy(buf, "abcd", 4);
DebugLog(buf); // abcd

// Lexicographical comparison
DebugLog(std::char_traits<char>::compare("abcd", "efgh",
4)); // -1
```

`std::fpos` is then build on `std::char_traits` to represent a position in a file. Usually we use the provided type aliases:

Type Alias	Character Traits
`std::streampos`	`std::char_traits<char>`
`std::wstreampos`	`std::char_traits<wchar_t>`
`std::u8streampos`	`std::char_traits<char8_t>`
`std::u16streampos`	`std::char_traits<char16_t>`

Type Alias	Character Traits
`std::u32streampos`	`std::char_traits<char32_t>`

Finally, there are some free functions that set flags on `std::ios_base` objects as an alternative to the `setf` member function:

```
#include <ios>

void Goo(std::ios_base& base)
{
    // Use strings like "true" or numbers like 1 for
bools
    std::boolalpha(base);
    std::noboolalpha(base);

    // Use uppercase or lowercase in hexadecimal numbers
and floats
    std::uppercase(base);
    std::nouppercase(base);
}
```

streambuf

The `<streambuf>` header provides just one class: `std::basic_streambuf`. This is an abstract base class of a way to input and output characters. It's meant to have its `virtual` functions overridden by derived classes in such a way that they implement the actual reading and writing from the stream. This might mean access to a network socket, file system, GPU memory, or any other place that serialized data can be transmitted to and received from.

We don't usually have a need to derive from this class. Instead, we typically use derivations that are already provided by the C++ Standard Library. These include "standard output, "standard error," "standard input," and access the file system. We'll see these when we look at `<iostream>` and `<fstream>`.

ostream and iostream

The `<ostream>` header provides output streams via the `std::basic_ostream` class template. There are two aliases for this that we typically use: `std::ostream` which aliases `std::basic_ostream<char>` and `std::wostream` for `wchar_t`.

we need to pass a `std::basic_streambuf` to construct a `std::basic_ostream`. This is also not commonly done. Instead, we typically use an already-created `std::basic_ostream` object. The `<iostream>` header provides a few pairs of these:

Global Object	Use	C# Equivalent
`std::cout`	Standard output of `char`	`Console.OpenStandardOutput`
`std::wcout`	Standard output of `wchar_t`	`Console.OpenStandardOutput`
`std::clog`	Standard error of `char`	`Console.OpenStandardError`
`std::wclog`	Standard error of `wchar_t`	`Console.OpenStandardError`
`std::cerr`	Unbuffered standard error of `char`	`Console.OpenStandardError`
`std::wcerr`	Unbuffered standard error of `wchar_t`	`Console.OpenStandardError`

Regardless of the object we choose to use, we have a few options for outputting data. The most typical is “formatted output” via the overloaded `<<` operator. This leads to the canonical “Hello, world!” application for C++:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

The `<<` operator is overloaded with all of the primitive types like `long`, `float`, `char`, `char*` (a C-string), and `bool`. It's common for us to add an overload for our own types so we can format them for output:

```
#include <iostream>

// Our own type
struct Point2
{
    float X;
    float Y;
};

// Overload basic_ostream's << operator for our own type
template <typename TChar>
std::basic_ostream<TChar>& operator<<(
    std::basic_ostream<TChar>& stream,
    const Point2& point)
{
    // Use the overloaded << operator with already-
supported primitive types
```

```
    stream << '(' << point.X << ", " << point.Y << ')';

    // Return the stream for operator chaining
    return stream;
}

// Print our own type to standard output
Point2 p{ 2, 4 };
std::cout << p << '\n'; // (2, 4)\n
```

At long last, we can write the `DebugLog` function! With the support of variadic templates, template specialization, and type-aware formatted output to a `std::basic_ostream`, it's actually only about 9 lines of code:

```
#include <iostream>

// Logging nothing just prints an empty line
void DebugLog()
{
    std::cout << '\n';
}

// Logging one value. This is the base case.
template <typename T>
void DebugLog(const T& val)
{
    std::cout << val << '\n';
```

```
}

// Logging two or more values
template <typename TFirst, typename TSecond, typename
...TRemain>
void DebugLog(const TFirst& first, const TSecond& second,
TRemain... remain)
{
    // Log the first value
    std::cout << first << ", ";

    // Recurse with the second value and any remaining
values
    DebugLog(second, remain...);
}

// Call the first function to print an empty line
DebugLog(); // \n

// Call the second function to print a single value
DebugLog('a'); // a\n

// Call the third function
// It prints "b, "
// It recurses with (1, true, "hello")
// It prints "1, "
// It recurses with (true, "hello")
// It prints "true, "
```

```
// It calls the second function with "hello"
// The second function prints "hello\n"
DebugLog('b', 1, true, "hello"); // b, 1, true, hello\n
```

Besides formatted output, there's also unformatted output for when we want to write raw data to an output stream. This data can be either a single character or a block of characters. We typically use this for outputting binary data while formatted output is typically used for strings and other human-readable data:

```
#include <iostream>

// Unformatted output of a single character
std::cout.put('a');

// Unformatted output of a block of characters
char buf[8];
for (int i = 0; i < sizeof(buf); ++i)
{
    buf[i] = 'a' + i;
}
std::cout.write(buf, sizeof(buf)); // abcdefgh
```

There are also functions for querying and controlling the position in the output stream. This has no meaning for `std::cout`, but makes sense for other output streams such as to files:

```
#include <iostream>
```

```
// Write a null byte at a position then restore the
position
void WriteNullAt(std::ostream& stream, std::streampos
pos)
{
    // Get stream position
    std::streampos oldPos{ stream.tellp() };

    // Seek stream position
    stream.seekp(pos);

    // Write the null byte
    stream.put(0);

    // Seek the stream back
    stream.seekp(oldPos);
}
```

An output stream that's buffered can also be explicitly flushed using, well, `flush`:

```
std::cout.flush();
```

`<ostream>` also provides a few "manipulator" functions. These are functions that, when passed to `<<` for formatted output, are called with the stream to determine what to output. We typically use them like this:

```cpp
#include <iostream>

// endl prints a "\n" character then calls flush()
std::cout << "Hello, world!" << std::endl;

// ends prints a null character, i.e. the value 0
std::cout << "Hello, world!" << std::ends;
```

istream

The `<istream>` header provides the opposite of `<ostream>`: support for input streams. The `std::basic_istream` class and its `std::istream` and `std::wistream` aliases make this possible. There's also a `std::basic_iostream` for streams that can input *and* output along with `std::iostream` and `std::wiostream` aliases. The `<iostream>` header provides `std::cin` and `std::wcin` global objects to read from "standard input."

As with output, we have both "formatted" and "unformatted" reading options. The "formatted" option enables the classic command line application to implement a basic calculator using the overloaded `>>` operator:

```
#include <iostream>

int main()
{
    std::cout << "Enter x:" << std::endl;
    int x;
    std::cin >> x;

    std::cout << "Enter y:" << std::endl;
    int y;
    std::cin >> y;

    std::cout << "x + y is " << (x+y) << std::endl;
}
```

Entering in some test values when prompted, we get the following output:

```
Enter x:
2
Enter y:
4
x + y is 6
```

We also have several options for unformatted input:

```
#include <iostream>

// Read 3 characters then print them
char buf[4] = { 0 };
std::cin.read(buf, 3); // Enter "abc"
DebugLog(buf); // abc

//// Read 1 character and ignore it
std::cin.ignore(1);

// Read until a character is found or the end of the
buffer is hit
std::cin.getline(buf, sizeof(buf), ';'); // Enter "ab;c"
DebugLog(buf); // ab
std::cin.getline(buf, sizeof(buf), ';'); // Enter
"abcdefg"
DebugLog(buf); // abc
```

```
// Put a character into the input stream
std::cin.putback('a');
std::cin.read(buf, 1);
DebugLog(buf); // a
```

iomanip

The `<iomanip>` header is full of “manipulator” functions that we can pass to formatted read and write operations. Here’s a sampling of the options:

```
#include <iomanip>
#include <iostream>
#include <numbers>

using namespace std;

// Output 255 as hexadecimal
cout << setbase(16) << 255 << endl; // ff

// Output pi with 3 digits of precision (whole and
fractional)
cout << setprecision(3) << numbers::pi << endl; // 3.14

// Set the width of the output and how it's filled.
Useful for columns.
auto row = [](auto num, auto name, char fill = ' ') {
    cout << '|' << setw(10) << setfill(fill) << name <<
'|';
    cout << setw(10) << setfill(fill) << num << '|' <<
endl;
};
row("Number", "Name");
```

```
row('-', '-', '-');
row(1, "One");
row(2, "Two");
// Prints:
// |    Number|      Name|
// |----------|----------|
// |         1|       One|
// |         2|       Two|

// Output cents as US Dollars
cout.imbue(locale("en_US"));
cout << std::showbase << put_money(250) << endl; // $2.50
```

fstream

The `<fstream>` header has facilities for file system I/O. At the lowest level, we have `std::basic_filebuf` which is a `std::basic_streambuf` that we can use for raw file system access. More typically, we use the `std::basic_ifstream`, `std::basic_ofstream`, and `std::basic_fstream` classes for input, output, and both. Aliases such as `std::fstream` are provided and most commonly seen. These are the rough equivalent of `FileStream` in C#:

```
#include <fstream>

void Foo()
{
    // Open the file for writing
    std::fstream stream{ "/path/to/file",
std::ios_base::out };

    // Formatted write to the file, including a flush via
endl
    stream << "hello" << std::endl;
} // fstream's destructor closes the file
```

As a derivative of `basic_iostream`, `basic_fstream` inherits all of its functionality. This includes the formatted and unformatted I/O functions such as the overloaded `<<` operator seen above. Besides this, a few file-specific member functions are on offer:

```
#include <fstream>

void Foo()
{
    // Open the file for writing
    std::fstream stream{ "/path/to/file",
std::ios_base::out };

    // Check if the file is open
    DebugLog(stream.is_open()); // true

    // Explicitly close the file without waiting for the
destructor
    stream.close();
    DebugLog(stream.is_open()); // false

    // Explicitly open a file without creating a new
stream
    stream.open("/path/to/other/file",
std::ios_base::out);
} // fstream's destructor closes any open file
```

sstream

As C# has `StringBuilder`, C++ has `std::basic_ostringstream` in the `<sstream>` header. This class template, typically aliased as `std::ostringstream`, allows writing to a string via the stream API:

```
#include <sstream>

// Create a stream for an empty string
std::ostringstream stream{};

// Formatted writing
stream << "Hello" << 123;

// Unformatted writing
stream.write("Goodbye", 8);

// Get a string for what was written
std::string str{ stream.str() };
DebugLog(str); // Hello123Goodbye
```

There's also an input version that reads from strings:

```
#include <sstream>

// Create a stream for a string
std::istringstream stream{ "Hello 123Goodbye" };
```

```
// Formatted reading
std::string str;
int num;
stream >> str >> num;
DebugLog(str); // Hello
DebugLog(num); // 123

// Unformatted reading
char buf[8] = { 0 };
stream.read(buf, 8);
DebugLog(buf); // Goodbye
```

And there's a combined `std::stringstream` that can both read and write:

```
#include <sstream>

// Create a stream for an empty string
std::stringstream stream{};

// Formatted writing
stream << "Hello 123";

// Change read position to the beginning
stream.seekg(std::ios_base::beg, 0);

// Formatted reading
std::string str;
```

```
int num;
stream >> str >> num;
DebugLog(str); // Hello
DebugLog(num); // 123
```

syncstream

The final header of the I/O library was introduced with C++20: `<syncstream>`. It provides `std:: basic_syncbuf` and `std::basic_osyncstream` to synchronize the writing to a stream from multiple threads. One motivating example is printing logs to standard output. Consider how this works without synchronization:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <functional>

// Prints "helloworld" to standard output 100 times
void Print(std::ostream& stream)
{
    for (int i = 0; i < 100; ++i)
    {
        stream << "helloworld" << std::endl;

std::this_thread::sleep_for(std::chrono::microseconds{ 1
});
    }
}

// Spawn a thread to print
std::jthread t{ Print, std::ref(std::cout) };
```

```
// Print on the main thread while the thread is running
Print(std::cout);
```

The exact output depends on OS scheduling, but this is likely to produce errors due to contention for the output stream and its internal buffer:

```
helloworld
helloworld
helloworldhelloworld

helloworld
helloworld
helloworld
helloworld
```

Here one of the threads printed `helloworld` but the other thread interrupted to print `helloworld\n` before the first thread could print its `\n` character. When the first thread resumed execution, it printed that `\n` resulting in two `\n` in a row: `\n\n`.

To avoid this problem, or any contention due to multi-threaded writing to a shared stream, we can use `std::basic_osyncstream` or its `std::osyncstream` alias to synchronize the writes:

```
#include <syncstream>
#include <iostream>
#include <thread>
#include <chrono>
```

```
#include <functional>

void Print(std::ostream& stream)
{
    for (int i = 0; i < 100; ++i)
    {
        stream << "helloworld" << std::endl;

std::this_thread::sleep_for(std::chrono::microseconds{ 1
});
    }
}

// Create a synchronized stream backed by std::cout
std::osyncstream out{ std::cout };

// Print to the synchronized stream
std::jthread t{ Print, std::ref(out) };
Print(std::cout);
```

Conclusion

The C++ “I/O streams” library is far more powerful than basic functionality like `printf` found in the C Standard Library. It’s not nearly as error-prone since it makes use of the C++ type system rather than manually-entered “format strings.” It’s far more extensible since we can write our own format functions, manipulator functions, and stream types to read and write from whatever kind of device we encounter.

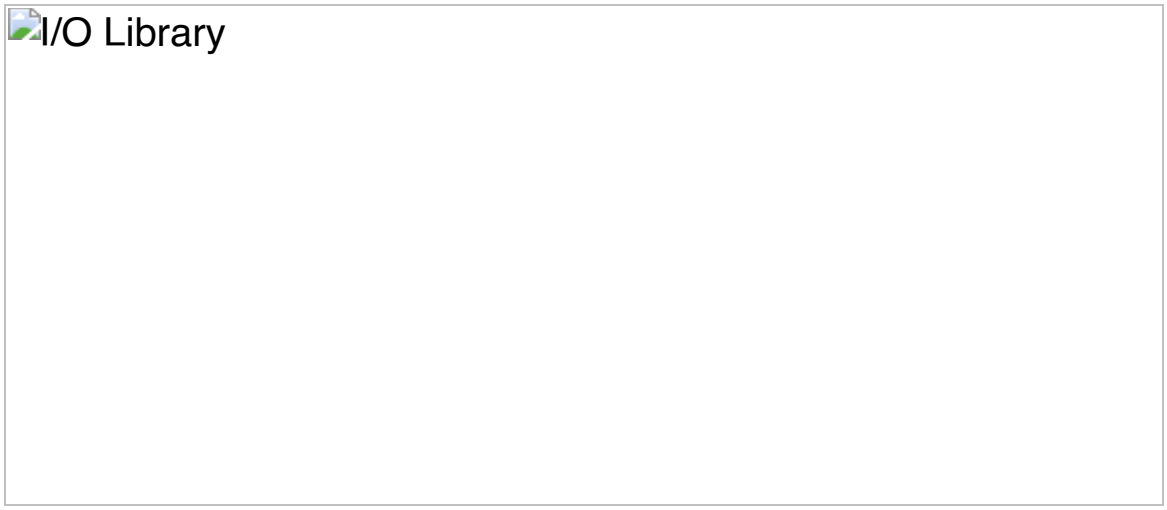
Compared to C#, its “unformatted” options are similar to `byte`-based options such as we find in the base `Stream` class. Its formatted options are similar to what we find in classes like `TextReader` and `TextWriter` except adapters like these aren’t required in C++. On the whole, the two libraries provide comparable functionality and even share the “stream” abstraction and terminology.

Perhaps the largest difference is in extensibility where C++ allows us to write our own types directly to a stream while C# typically requires us to allocate a `String` object from our `ToString` function. The addition of `std::osyncstream` in C++20 is also a nice addition as it saves us from multi-threaded synchronization of stream writes regardless of language.

50. I/O Library

Library Layout and iosfwd

The I/O library subset of the broader C++ Standard Library contains several header files that often `#include` each other. Here's how those relationships look:



The most basic usage of the I/O library is to `#include <iosfwd>`. This header provides "forward" declarations of I/O types. These can then be named, such as by pointer or reference types. They can't be used by value or by accessing any of their members. `<iosfwd>` really just exists to speed up compilation when I/O types only need to be named and their full definitions, which are slower to compile, aren't needed.

We'll see all the types in `<iosfwd>` as we go through each of the headers that `#include` it and then provide definitions.

ios

The `<ios>` header contains basic tools that the rest of the I/O library uses. First up, there's `std::ios_base` which serves as the abstract base class of all the I/O "stream" classes. We'll see what those classes look like soon, but suffice to say a "stream" is an abstract input and/or output that can be backed by a file, "standard output", a string, and so forth. It's very similar to the C# `Stream` abstract base class.

Here's some of what `std::ios_base` provides:

```
#include <ios>
#include <locale>

void Goo(std::ios_base& base)
{
    // Get the flags that control formatting
    std::ios_base::fmtflags f{ base.flags() };
    DebugLog((f & std::ios_base::dec) != 0); // Maybe
true
    DebugLog((f & std::ios_base::hex) != 0); // Maybe
false
    DebugLog((f & std::ios_base::boolalpha) != 0); //
Maybe false

    // Set and unset a format flag
    base.setf(std::ios_base::boolalpha);
    DebugLog((base.flags() & std::ios_base::boolalpha) !=
```

```
0); // true
    base.unsetf(std::ios_base::boolalpha);
    DebugLog((base.flags() & std::ios_base::boolalpha) !=
0); // false

    // Set and get floating-point precision
    base.precision(2);
    DebugLog(base.precision()); // 2

    // Set and get the minimum number of characters that
some operations print
    base.width(10);
    DebugLog(base.width()); // 10

    // Set and get the locale
    base.imbue(std::locale{ "de-DE" });
    DebugLog(base.getloc().name()); // de-DE

    try
    {
        throw std::ios_base::failure{ "some I/O error" };
    }
    catch (const std::ios_base::failure& ex)
    {
        DebugLog(ex.what()); // some I/O error
    }

    // Ways of opening streams
```

```
    // These are bit flags to form a mask
    std::ios_base::openmode mode{
        std::ios_base::app | // Append
        std::ios_base::binary | // Binary
        std::ios_base::in | // Read
        std::ios_base::out | // Write
        std::ios_base::trunc | // Overwrite
        std::ios_base::ate // Open at end of stream
    };

    // Bit flags forming the state of a stream
    std::ios_base::iostate state{
        std::ios_base::goodbit | // No error
        std::ios_base::badbit | // Unrecoverable error
        std::ios_base::failbit | // Operation failed
(e.g. formatting failed)
        std::ios_base::eofbit // End of stream
    };

    // Directions to seek
    std::ios_base::seekdir dir = std::ios_base::beg; //
Beginning of stream
    dir = std::ios_base::end; // End of stream
    dir = std::ios_base::cur; // From the current
position
}
```

There's also `std::char_traits`, which is a class template with static functions that provide functionality for operations on particular kinds of characters:

```
#include <ios>

// Single-character operations
DebugLog(std::char_traits<char>::eq('a', 'a')); // true
DebugLog(std::char_traits<char>::eof()); // -1

// Copy multiple characters
char buf[5];
std::char_traits<char>::copy(buf, "abcd", 4);
DebugLog(buf); // abcd

// Lexicographical comparison
DebugLog(std::char_traits<char>::compare("abcd", "efgh",
4)); // -1
```

`std::fpos` is then build on `std::char_traits` to represent a position in a file. Usually we use the provided type aliases:

Type Alias	Character Traits
`std::streampos`	`std::char_traits<char>`
`std::wstreampos`	`std::char_traits<wchar_t>`
`std::u8streampos`	`std::char_traits<char8_t>`
`std::u16streampos`	`std::char_traits<char16_t>`

Type Alias	Character Traits
`std::u32streampos`	`std::char_traits<char32_t>`

Finally, there are some free functions that set flags on `std::ios_base` objects as an alternative to the `setf` member function:

```
#include <ios>

void Goo(std::ios_base& base)
{
    // Use strings like "true" or numbers like 1 for
bools
    std::boolalpha(base);
    std::noboolalpha(base);

    // Use uppercase or lowercase in hexadecimal numbers
and floats
    std::uppercase(base);
    std::nouppercase(base);
}
```

streambuf

The `<streambuf>` header provides just one class: `std::basic_streambuf`. This is an abstract base class of a way to input and output characters. It's meant to have its `virtual` functions overridden by derived classes in such a way that they implement the actual reading and writing from the stream. This might mean access to a network socket, file system, GPU memory, or any other place that serialized data can be transmitted to and received from.

We don't usually have a need to derive from this class. Instead, we typically use derivations that are already provided by the C++ Standard Library. These include "standard output, "standard error," "standard input," and access the file system. We'll see these when we look at `<iostream>` and `<fstream>`.

ostream and iostream

The `<ostream>` header provides output streams via the `std::basic_ostream` class template. There are two aliases for this that we typically use: `std::ostream` which aliases `std::basic_ostream<char>` and `std::wostream` for `wchar_t`.

we need to pass a `std::basic_streambuf` to construct a `std::basic_ostream`. This is also not commonly done. Instead, we typically use an already-created `std::basic_ostream` object. The `<iostream>` header provides a few pairs of these:

Global Object	Use	C# Equivalent
`std::cout`	Standard output of `char`	`Console.OpenStandardOutput`
`std::wcout`	Standard output of `wchar_t`	`Console.OpenStandardOutput`
`std::clog`	Standard error of `char`	`Console.OpenStandardError`
`std::wclog`	Standard error of `wchar_t`	`Console.OpenStandardError`
`std::cerr`	Unbuffered standard error of `char`	`Console.OpenStandardError`
`std::wcerr`	Unbuffered standard error of `wchar_t`	`Console.OpenStandardError`

Regardless of the object we choose to use, we have a few options for outputting data. The most typical is “formatted output” via the overloaded `<<` operator. This leads to the canonical “Hello, world!” application for C++:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

The `<<` operator is overloaded with all of the primitive types like `long`, `float`, `char`, `char*` (a C-string), and `bool`. It's common for us to add an overload for our own types so we can format them for output:

```
#include <iostream>

// Our own type
struct Point2
{
    float X;
    float Y;
};

// Overload basic_ostream's << operator for our own type
template <typename TChar>
std::basic_ostream<TChar>& operator<<(
    std::basic_ostream<TChar>& stream,
    const Point2& point)
{
    // Use the overloaded << operator with already-
supported primitive types
```

```
    stream << '(' << point.X << ", " << point.Y << ')';

    // Return the stream for operator chaining
    return stream;
}

// Print our own type to standard output
Point2 p{ 2, 4 };
std::cout << p << '\n'; // (2, 4)\n
```

At long last, we can write the `DebugLog` function! With the support of variadic templates, template specialization, and type-aware formatted output to a `std::basic_ostream`, it's actually only about 9 lines of code:

```
#include <iostream>

// Logging nothing just prints an empty line
void DebugLog()
{
    std::cout << '\n';
}

// Logging one value. This is the base case.
template <typename T>
void DebugLog(const T& val)
{
    std::cout << val << '\n';
```

```
}

// Logging two or more values
template <typename TFirst, typename TSecond, typename
...TRemain>
void DebugLog(const TFirst& first, const TSecond& second,
TRemain... remain)
{
    // Log the first value
    std::cout << first << ", ";

    // Recurse with the second value and any remaining
values
    DebugLog(second, remain...);
}

// Call the first function to print an empty line
DebugLog(); // \n

// Call the second function to print a single value
DebugLog('a'); // a\n

// Call the third function
// It prints "b, "
// It recurses with (1, true, "hello")
// It prints "1, "
// It recurses with (true, "hello")
// It prints "true, "
```

```
// It calls the second function with "hello"
// The second function prints "hello\n"
DebugLog('b', 1, true, "hello"); // b, 1, true, hello\n
```

Besides formatted output, there's also unformatted output for when we want to write raw data to an output stream. This data can be either a single character or a block of characters. We typically use this for outputting binary data while formatted output is typically used for strings and other human-readable data:

```
#include <iostream>

// Unformatted output of a single character
std::cout.put('a');

// Unformatted output of a block of characters
char buf[8];
for (int i = 0; i < sizeof(buf); ++i)
{
    buf[i] = 'a' + i;
}
std::cout.write(buf, sizeof(buf)); // abcdefgh
```

There are also functions for querying and controlling the position in the output stream. This has no meaning for `std::cout`, but makes sense for other output streams such as to files:

```
#include <iostream>
```

```
// Write a null byte at a position then restore the
position
void WriteNullAt(std::ostream& stream, std::streampos
pos)
{
    // Get stream position
    std::streampos oldPos{ stream.tellp() };

    // Seek stream position
    stream.seekp(pos);

    // Write the null byte
    stream.put(0);

    // Seek the stream back
    stream.seekp(oldPos);
}
```

An output stream that's buffered can also be explicitly flushed using, well, `flush`:

```
std::cout.flush();
```

`<ostream>` also provides a few "manipulator" functions. These are functions that, when passed to `<<` for formatted output, are called with the stream to determine what to output. We typically use them like this:

```cpp
#include <iostream>

// endl prints a "\n" character then calls flush()
std::cout << "Hello, world!" << std::endl;

// ends prints a null character, i.e. the value 0
std::cout << "Hello, world!" << std::ends;
```

istream

The `<istream>` header provides the opposite of `<ostream>`: support for input streams. The `std::basic_istream` class and its `std::istream` and `std::wistream` aliases make this possible. There's also a `std::basic_iostream` for streams that can input *and* output along with `std::iostream` and `std::wiostream` aliases. The `<iostream>` header provides `std::cin` and `std::wcin` global objects to read from "standard input."

As with output, we have both "formatted" and "unformatted" reading options. The "formatted" option enables the classic command line application to implement a basic calculator using the overloaded `>>` operator:

```
#include <iostream>

int main()
{
    std::cout << "Enter x:" << std::endl;
    int x;
    std::cin >> x;

    std::cout << "Enter y:" << std::endl;
    int y;
    std::cin >> y;

    std::cout << "x + y is " << (x+y) << std::endl;
}
```

Entering in some test values when prompted, we get the following output:

```
Enter x:
2
Enter y:
4
x + y is 6
```

We also have several options for unformatted input:

```
#include <iostream>

// Read 3 characters then print them
char buf[4] = { 0 };
std::cin.read(buf, 3); // Enter "abc"
DebugLog(buf); // abc

//// Read 1 character and ignore it
std::cin.ignore(1);

// Read until a character is found or the end of the
buffer is hit
std::cin.getline(buf, sizeof(buf), ';'); // Enter "ab;c"
DebugLog(buf); // ab
std::cin.getline(buf, sizeof(buf), ';'); // Enter
"abcdefg"
DebugLog(buf); // abc
```

```
// Put a character into the input stream
std::cin.putback('a');
std::cin.read(buf, 1);
DebugLog(buf); // a
```

iomanip

The `<iomanip>` header is full of “manipulator” functions that we can pass to formatted read and write operations. Here’s a sampling of the options:

```
#include <iomanip>
#include <iostream>
#include <numbers>

using namespace std;

// Output 255 as hexadecimal
cout << setbase(16) << 255 << endl; // ff

// Output pi with 3 digits of precision (whole and
fractional)
cout << setprecision(3) << numbers::pi << endl; // 3.14

// Set the width of the output and how it's filled.
Useful for columns.
auto row = [](auto num, auto name, char fill = ' ') {
    cout << '|' << setw(10) << setfill(fill) << name <<
'|';
    cout << setw(10) << setfill(fill) << num << '|' <<
endl;
};
row("Number", "Name");
```

```
row('-', '-', '-');
row(1, "One");
row(2, "Two");
// Prints:
// |    Number|      Name|
// |----------|----------|
// |         1|       One|
// |         2|       Two|

// Output cents as US Dollars
cout.imbue(locale("en_US"));
cout << std::showbase << put_money(250) << endl; // $2.50
```

fstream

The `<fstream>` header has facilities for file system I/O. At the lowest level, we have `std::basic_filebuf` which is a `std::basic_streambuf` that we can use for raw file system access. More typically, we use the `std::basic_ifstream`, `std::basic_ofstream`, and `std::basic_fstream` classes for input, output, and both. Aliases such as `std::fstream` are provided and most commonly seen. These are the rough equivalent of `FileStream` in C#:

```
#include <fstream>

void Foo()
{
    // Open the file for writing
    std::fstream stream{ "/path/to/file",
std::ios_base::out };

    // Formatted write to the file, including a flush via
endl
    stream << "hello" << std::endl;
} // fstream's destructor closes the file
```

As a derivative of `basic_iostream`, `basic_fstream` inherits all of its functionality. This includes the formatted and unformatted I/O functions such as the overloaded `<<` operator seen above. Besides this, a few file-specific member functions are on offer:

```
#include <fstream>

void Foo()
{
    // Open the file for writing
    std::fstream stream{ "/path/to/file",
std::ios_base::out };

    // Check if the file is open
    DebugLog(stream.is_open()); // true

    // Explicitly close the file without waiting for the
destructor
    stream.close();
    DebugLog(stream.is_open()); // false

    // Explicitly open a file without creating a new
stream
    stream.open("/path/to/other/file",
std::ios_base::out);
} // fstream's destructor closes any open file
```

sstream

As C# has `StringBuilder`, C++ has `std::basic_ostringstream` in the `<sstream>` header. This class template, typically aliased as `std::ostringstream`, allows writing to a string via the stream API:

```
#include <sstream>

// Create a stream for an empty string
std::ostringstream stream{};

// Formatted writing
stream << "Hello" << 123;

// Unformatted writing
stream.write("Goodbye", 8);

// Get a string for what was written
std::string str{ stream.str() };
DebugLog(str); // Hello123Goodbye
```

There's also an input version that reads from strings:

```
#include <sstream>

// Create a stream for a string
std::istringstream stream{ "Hello 123Goodbye" };
```

```
// Formatted reading
std::string str;
int num;
stream >> str >> num;
DebugLog(str); // Hello
DebugLog(num); // 123

// Unformatted reading
char buf[8] = { 0 };
stream.read(buf, 8);
DebugLog(buf); // Goodbye
```

And there's a combined `std::stringstream` that can both read and write:

```
#include <sstream>

// Create a stream for an empty string
std::stringstream stream{};

// Formatted writing
stream << "Hello 123";

// Change read position to the beginning
stream.seekg(std::ios_base::beg, 0);

// Formatted reading
std::string str;
```

```
int num;
stream >> str >> num;
DebugLog(str); // Hello
DebugLog(num); // 123
```

syncstream

The final header of the I/O library was introduced with C++20: `<syncstream>`. It provides `std:: basic_syncbuf` and `std::basic_osyncstream` to synchronize the writing to a stream from multiple threads. One motivating example is printing logs to standard output. Consider how this works without synchronization:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <functional>

// Prints "helloworld" to standard output 100 times
void Print(std::ostream& stream)
{
    for (int i = 0; i < 100; ++i)
    {
        stream << "helloworld" << std::endl;

std::this_thread::sleep_for(std::chrono::microseconds{ 1
});
    }
}

// Spawn a thread to print
std::jthread t{ Print, std::ref(std::cout) };
```

```
// Print on the main thread while the thread is running
Print(std::cout);
```

The exact output depends on OS scheduling, but this is likely to produce errors due to contention for the output stream and its internal buffer:

```
helloworld
helloworld
helloworldhelloworld

helloworld
helloworld
helloworld
helloworld
```

Here one of the threads printed `helloworld` but the other thread interrupted to print `helloworld\n` before the first thread could print its `\n` character. When the first thread resumed execution, it printed that `\n` resulting in two `\n` in a row: `\n\n`.

To avoid this problem, or any contention due to multi-threaded writing to a shared stream, we can use `std::basic_osyncstream` or its `std::osyncstream` alias to synchronize the writes:

```
#include <syncstream>
#include <iostream>
#include <thread>
#include <chrono>
```

```
#include <functional>

void Print(std::ostream& stream)
{
    for (int i = 0; i < 100; ++i)
    {
        stream << "helloworld" << std::endl;

std::this_thread::sleep_for(std::chrono::microseconds{ 1
});
    }
}

// Create a synchronized stream backed by std::cout
std::osyncstream out{ std::cout };

// Print to the synchronized stream
std::jthread t{ Print, std::ref(out) };
Print(std::cout);
```

Conclusion

The C++ “I/O streams” library is far more powerful than basic functionality like `printf` found in the C Standard Library. It’s not nearly as error-prone since it makes use of the C++ type system rather than manually-entered “format strings.” It’s far more extensible since we can write our own format functions, manipulator functions, and stream types to read and write from whatever kind of device we encounter.

Compared to C#, its “unformatted” options are similar to `byte`-based options such as we find in the base `Stream` class. Its formatted options are similar to what we find in classes like `TextReader` and `TextWriter` except adapters like these aren’t required in C++. On the whole, the two libraries provide comparable functionality and even share the “stream” abstraction and terminology.

Perhaps the largest difference is in extensibility where C++ allows us to write our own types directly to a stream while C# typically requires us to allocate a `String` object from our `ToString` function. The addition of `std::osyncstream` in C++20 is also a nice addition as it saves us from multi-threaded synchronization of stream writes regardless of language.

51. Missing Library Features

Overview

The .NET library for C# is truly gigantic. It covers an extremely wide-ranging set of functionality for a programming language's standard library. Even other "batteries included" languages like Python and Java have nowhere near as much functionality as .NET does. It absolutely dwarfs the C++ Standard Library, which has historically limited itself to truly core functionality that can be applied to virtually any computing device, is extremely mature, and can be achieve consensus in a large, diverse standards committee. .NET, being controlled by Microsoft, includes a great many features that are specific to Microsoft Windows, the platforms it happens to run on, and the technologies it encourages using.

As a result of this, C# developers have a ton of general-purpose tools such as for JSON serialization but also a ton of tools such as access to GDI+ that will most likely not apply to any code we write. We can ignore them, but it makes providing an implementation of .NET on non-Windows platforms more difficult and therefore less likely to be accomplished and available. .NET itself has fractured into a collection of overlapping libraries which may or may not be available on any given platform: .NET Framework, .NET Core, .NET Compact Framework, .NET Micro Framework, Microsoft Silverlight, Mono, and Unity. Some of these are deprecated and others have been renamed but there are still broad gulfs between libraries like .NET Core and Unity.

Like the C++ language itself, the C++ Standard Library is a *standard*. There are various implementations of the standard, but they all have approximately the same features. Variance between them is mostly in the form of unspecified behavior such as exception message strings and deviances from the standard such as adding or removing

some (usually relatively-minor) features. This is especially true in newly-released standards such as C++20 at the time of writing.

In a way the comparison between the two languages' standard libraries is a bit apples-to-oranges, but in this chapter we'll look at some of the high-level sorts of functionality a C# developer using one of the .NET libraries won't find in the C++ Standard Library.

Cryptography

The `System.Security.Cryptography` namespace provides C# programmers with access to many common cryptographic algorithms including AES, RSA, and the SHA family. The `System.Net.Security` namespace provides TLS and SSL.

The C++ Standard Library doesn't have any cryptography functionality built in, so we seek other libraries to provide the needed functionality. Thankfully, we can make easy use of C libraries as well as C++ libraries so we have many options. Here are a few of them:

Library	Language	License	Crypto Algorithms	TLS and SSL
Botan	C++	BSD	Yes	Yes
OpenSSL	C	Apache	Yes	Yes
Crypto++	C++	Boost	Yes	No
libsodium	C	ISC	Yes	No

Compression

In C#, we use the `System.IO.Compression` to access compression algorithms such as GZip and Deflate as well as ZIP archives. Again, we turn to non-standard libraries when we need compression functionality with C++. Here are a few:

Library	Language	License	Algorithms
zlib	C	zlib	GZip, Deflate
Zipper	C++	MIT	ZIP
LZ4	C	BSD	LZ4
LZMA SDK	C and C++	Public Domain	LZMA, LZMA2

Networking

`System.Net` provides access to low-level sockets and WebSockets while sub-namespaces provide higher-level functionality. For example, `System.Net.Http` provides HTTP functionality such as a client: `HttpClient`.

Proposals have been made to add networking functionality to the C++ Standard Library, but so far haven't been accepted. While we wait for standardization, we can make use of many existing libraries including these:

Library	Language	Licence	Protocols
Boost.Asio	C++	Boost	TCP, UDP, ICMP, serial ports, UNIX sockets, Windows `HANDLE`, SSL (via OpenSSL)
Boost.Beast	C++	Boost	Via Boost.Asio and OpenSSL: HTTP (client and server), WebSocket (client and server)
cpp-httplib	C++	MIT	HTTP (client and server)
WebSocket++	C++	BSD	WebSocket

Graphical User Interfaces

.NET has built-in support for three Windows GUIs: GDI+ in `System.Drawing`, Windows Forms in `System.Windows.Forms`, and Windows Presentation Foundation in `System.Windows`. While not part of .NET, Microsoft makes Xamarin available for cross-platform GUI development on Windows, macOS, Linux, Android, and iOS but not web browsers.

C++ has no built-in GUI support, but can access Windows Forms and Windows Presentation Foundation via C++/CLI. Quite a few libraries are also available for cross-platform GUI development:

Library	Language	Licence	Windows	macOS	Linux	Android	iOS	Web Browser
Qt	C++	LGPL, GPL, Commercial	Yes	Yes	Yes	Yes	Yes	Yes
GTK+	C	LGPL	Yes	Yes	Yes			
wxWidgets	C++	wxWindows	Yes	Yes	Yes			
Dear ImGui	C++	MIT	Yes	Yes	Yes	Yes	Yes	Yes

CPU Intrinsics

`System.Runtime.Intrinsics` and its sub-namespaces `System.Runtime.Intrinsics.X86` and `System.Runtime.Intrinsics.Arm` contain “intrinsics” for x86 and ARM CPUs. These are functions whose calls are translated by the compiler directly into a named CPU instruction. They provide low-level control without needing to resort to assembly code.

C++ doesn’t have standardized intrinsics, but they’re widely available:

- Microsoft Visual Studio
- Intel Intrinsics Guide
- GCC Builtins including intrinsics
- ARM NEON Intrinsics

JSON

The JSON format is supported directly by .NET in the `System.Text.Json` namespace. The C++ Standard Library doesn't directly support this format or any others, so we instead make use of libraries:

Library	Language	License
JSON for Modern C++	C++	MIT
RapidJSON	C++	MIT
JsonCpp	C++	MIT
Boost.JSON	C++	Boost

Debugging

The `System.Diagnostics` namespace in .NET provides some useful debugging features. For example, we can use `Debugger.Break` to break an interactive debugger. There's also the `StackTrace` class to get stack traces, especially when we run into problems.

C++ doesn't provide either of these in its Standard Library, but we can still access them. To break an interactive debugger on Windows, we `#include <debugapi.h>` and call the DebugBreak function. On UNIX systems, we `#include <signal.h>` and call the raise function with `SIGTRAP` as the argument.

Support for stack traces has been proposed for the Standard Library, but we'll have to wait until at least C++23 for it to be adopted. For now, the Boost-licensed Boost.Stacktrace library that the Standard Library proposal is based on is available to fill the gap.

Database Clients

The `System.Data` goes as far as to build in support for particular databases. `System.Data.SqlClient` is a client for Microsoft SQL Server and `System.Data.OracleClient` is a client for Oracle Database.

The C++ Standard Library never endorses particular software products like these, so it has zero support for databases. Instead, Database vendors typically provide a C++ client, connector, or driver:

Library	Language	License	Database
ODBC Driver for Microsoft SQL Server	C	Commercial	Microsof SQL Server
Oracle C++ Call Interface	C+++	Commercial	Oracle Database
MySQL Connector	C++	GPL or Commercial	MySQL
MongoDB Driver	C++	Apache	MongoDB

Conclusion

The .NET family of standard libraries for C# often take a different design approach to the C++ Standard Library. They're much more comfortable building in support for particular software such as Oracle Database. They'll add OS-specific functionality such as to build GDI+ user interfaces. They'll also add CPU intrinsics that are specific to particular processor architectures such as x86 and ARM.

The C++ Standard Library is designed in a more abstract way. It doesn't choose to support any particular database, OS, processor, algorithm, or data format. We'll simply have to look outside of the standard if we want any functionality related to a concrete software or hardware product.

The area of overlap between .NET and the C++ Standard Library relates to general tools such as collections and file system access. Sometimes the C++ Standard Library has more available here, such as with its doubly-ended queue type. Other times .NET has more available, such as with its ability to break an interactive debugger or open a network socket. These tools *could* be added to the C++ Standard Library in the future, but for now we need to employ other libraries to get access to them.

52. Idioms and Best Practices

Guides

There are several existing, popular guides that aim to impose programming standards on C++ codebases for a variety of reasons. These reasons range from trivialities such as formatting to standardization of error-handling and outright bans on certain language features. Additionally, many teams and organizations will create their own in-house rules and possibly enforce them with tools like ClangFormat.

Here are some of the most popular public guides:

- C++ Core Guidelines: maintained by the original creator of C++ and a top C++ standards committee member for general-purpose code rather than a particular industry, domain, or product. The Guidelines Support Library (GSL) by Microsoft and related tools in Visual Studio support these guidelines.
- Google C++ Style Guide: used by Google for their massive C++ codebase spanning the web, mobile devices, and more.
- Unreal Engine Coding Standard: a much more brief guide focused specifically on C++ written for Unreal Engine games

There are strong disagreements between these guides and even their scopes vary widely. This chapter is mostly about areas that have some semblance of agreement in the broader community of C++ developers. It's not an attempt to create a new guide.

Use macros extremely rarely

Macros are evaluated at an early stage of compilation and essentially operate as scope-unaware text-replacers that aren't fully compatible with either variables or functions. They're notoriously difficult to debug and even understand. While they're essential to implement assertions, there are very few other cases where they are preferable to regular C++ code:

```
// Avoid
#define PI 3.14

// Encourage
const float PI = 3.14;
```

```
// Avoid
#define SQUARE(x) x * x

// Encourage
float Square(float x)
{
    return x * x;
}
```

Add include guards to every header

As of this writing, modules are not yet in common usage. While still using the classic header file-based build system, every header file should have “include guards” to prevent redundant definitions by multiple `#include` directives:

```
// Avoid
struct Point2
{
    float X;
    float Y;
};

// Encourage
#ifndef POINT2_HPP
#define POINT2_HPP
struct Point2
{
    float X;
    float Y;
};
#endif

// Encourage (non-standard but widely-supported
alternative)
#pragma once
struct Point2
```

```
{
    float X;
    float Y;
};
```

Include dependencies directly instead of relying on indirect includes

When one file uses code in another file, it should `#include` the file declaring that code directly rather than relying on another header to `#include` the desired code. This prevents compilation errors if the middle header removes its `#include`.

```
////////
// Avoid
////////

// a.h
struct A {};

// b.h
#include "a.h"
struct B : A {};

// c.h
#include "b.h"
A a; // Not in b.h

////////////
// Encourage
////////////

// a.h
```

```
struct A {};

// b.h
#include "a.h"
struct B : A {};

// c.h
#include "a.h"
A a;
```

Don’t call virtual functions in constructors

Virtual functions rely on a table that’s initialized by the constructors of the classes in an inheritance hierarchy. Calling `virtual` functions before these are set up can result in crashes or calling the wrong version of the function:

```
// Avoid
struct Parent
{
    Parent()
    {
        Foo();
    }

    virtual void Foo()
    {
        DebugLog("Parent");
    }
};
struct Child : Parent
{
    virtual void Foo() override
    {
        DebugLog("Child");
    }
};
Child c; // Prints "Parent" not "Child"!
```

```
// Encourage designs that do not require such calls
```

Don’t use variadic functions

Variadic functions aren’t type-safe, rely on error-prone macros, are difficult to optimize, and often result in error-prone APIs. They should be avoided in favor of techniques such as fold expressions or the use of container types:

```
// Avoid
void DebugLog(int count, ...)
{
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i)
    {
        const char* log = va_arg(args, const char*);
        std::cout << log << ", ";
    }
    va_end(args);
}
DebugLog(4, "foo", "bar", "baz"); // Whoops! 4 reads
beyond the last arg!

// Encourage
void DebugLog()
{
    std::cout << '\n';
}
template <typename T>
```

```
void DebugLog(const T& val)
{
    std::cout << val << '\n';
}
template <typename TFirst, typename TSecond, typename
...TRemain>
void DebugLog(const TFirst& first, const TSecond& second,
TRemain... remain)
{
    std::cout << first << ", ";
    DebugLog(second, remain...);
}
DebugLog("foo", "bar", "baz"); // No need for a count.
Can't get it wrong.
```

No naked new and delete

The `new` and `delete` operators to dynamically allocate memory should be a rare sight. Instead, “owning” types such as containers and smart pointers should call `new` in their constructors and `delete` in their destructors to ensure that memory is always cleaned up:

```
// Avoid
struct Game
{
    Stats* stats;
    Game()
        : stats{new Stats()}
    {
    }
    ~Game()
    {
        delete stats;
    }
};

// Encourage
struct Game
{
    std::unique_ptr<Stats> stats;
    Game()
        : stats{std::make_unique<Stats>()}
    {
```

```
    }
};
```

```
// Avoid
struct FloatBuffer
{
    float* floats;
    FloatBuffer(int32_t count)
        : floats{new float[count]}
    {
    }
    ~FloatBuffer()
    {
        delete [] floats;
    }
};

// Encourage
struct FloatBuffer
{
    std::vector<float> floats;
    FloatBuffer(int32_t count)
        : floats{count}
    {
    }
};
```

Prefer range-based loops

The most common loop is from the beginning to the end of a collection. To avoid mistakes and make this more terse, use a [range-based]) `for` loop instead of the three-part `for` loop, a `while` loop, or a `do-while` loop:

```
// Avoid
for (
    std::vector<float>::iterator it = floats.begin();
    it != floats.end();
    ++i)
{
    DebugLog(*it);
}

// Encourage
for (float f : floats)
{
    DebugLog(f);
}
```

Use scoped enums instead of unscoped enums

To avoid adding all the enumerators of an unscoped `enum` to the surrounding scope, use a scoped enumeration:

```
// Avoid
enum Colors { Red, Green, Blue };
uint32_t Red = 0x00ff00ff; // Error: redefinition because
Red escaped the enum

// Encourage
enum class Colors { Red, Green, Blue };
uint32_t Red = 0x00ff00ff; // OK
```

Don’t breach namespaces in headers

When commonly using the members of a namespace, it can be convenient to pull them out with `using namespace`. When this is done in a header, the files that `#include` it have this decision forced on them. This can lead to namespace collisions and confusion, so it should be avoided.

```
// Avoid
using namespace std;
struct Name
{
    string First;
    string Last;
};

// Encourage
struct Name
{
    std::string First;
    std::string Last;
};
```

Make single-parameter constructors explicit

Constructors default to allowing implicit conversion, which can be surprising and expensive. Use the `explicit` keyword to disallow this behavior:

```
// Avoid
struct Buffer
{
    std::vector<float> floats;
    Buffer(int x)
        : floats{x}
    {
    }
};
void DoStuff(Buffer b)
{
}
DoStuff(1'000'000); // Allocates a Buffer of one million floats!

// Encourage
struct Buffer
{
    std::vector<float> floats;
    explicit Buffer(int x)
        : floats{x}
    {
```

```
    }
};
void DoStuff(Buffer b)
{
}
DoStuff(1'000'000); // Compiler error
```

Don’t use C casts

C-style casts may just change the type but also might perform value conversion. They’re hard to search for as they blend in with other parentheses. Instead, use a named C++ cast for better control and easier searching:

```
// Avoid
const float val = 5.5;
int x = (int)val; // Changes type, truncates, and removes constness!
DebugLog(x);

// Encourage
const float val = 5.5;
int x = const_cast<int>(val); // Compiler error
DebugLog(x);
```

Use specific integer sizes

All the way back in the second chapter of the book, we learned that the size guarantees for primitive types like `long` are very weak. They should be avoided in favor of guaranteed sizes in `<cstdint>`:

```
// Avoid
long GetFileSize(const char* path)
{
    // Does this support files larger than 4 GB?
    // Depends on whether long is 32-bit or 64-bit
}

// Encourage
int64_t GetFileSize(const char* path)
{
    // Definitely supports large files
}
```

Use nullptr

`NULL` is an implementation-defined variable-like macro that even requires a `#include` to use. It's a null pointer constant, but of unknown type and erroneously usable in arithmetic. In contrast `nullptr` is not a macro or an integer, requires no header, and even has its own type which can be used in overload resolution. It should be used to represent null pointers:

```
// Avoid
int* p = NULL;

// Encourage
int* p = nullptr;
```

Follow the Rule of Zero

Most classes shouldn't need any copy constructors, move constructors, destructors, copy assignment operators, or move assignment operators. Instead, their data members should take care of these functions. This is called the "rule of zero" because no special functions need to be added. It's the simplest approach and the hardest to implement incorrectly:

```
// Avoid
struct Player
{
    std::string Name;
    int32_t Score;

    Player(std::string name, int32_t score)
        : Name{ name }
        , Score{ score }
    {
    }

    Player(const Player& other)
        : Name{ other.Name }
        , Score{ other.Score }
    {
    }

    Player(Player&& other) noexcept
        : Name{ std::move(other.Name) }
```

```
        , Score{ std::move(other.Score) }
    {
    }

    virtual ~Player()
    {
    }

    Player& operator=(const Player& other)
    {
        Name = other.Name;
        Score = other.Score;
        return *this;
    }

    Player& operator==(Player&& other)
    {
        Name = std::move(other.Name);
        Score = std::move(other.Score);
        return *this;
    }
};
Player p{ "Jackson", 1000 };

// Encourage
struct Player
{
    std::string Name;
```

```
    int32_t Score;
};
Player p{ "Jackson", 1000 };
```

Follow the Rule of Five

In cases where the Rule of Zero can't be followed and a special function needs to be added, add all five of them to handle all the ways that objects can be copied, moved, and destroyed:

```
// Avoid
struct File
{
    std::string Path;
    const char* Mode;
    FILE* Handle;

    File(std::string path, const char* mode)
        : Path{ path }
        , Mode{ mode }
        , Handle(fopen(path.c_str(), mode))
    {
    }

    File(const File& other)
        : Path{ other.Path }
        , Mode{ other.Mode }
        , Handle(fopen(other.Path.c_str(), other.Mode))
    {
    }

    virtual ~File()
```

```
    {
        if (Handle)
        {
            fclose(Handle);
        }
    }

    File& operator=(const File& other)
    {
        Path = other.Path;
        Mode = other.Mode;
        Handle = fopen(other.Path.c_str(), other.Mode);
        return *this;
    }

    // No move constructor or move assignment operator
    // Expensive copies will be required: more file open
and close operations!
};

// Encourage
struct File
{
    std::string Path;
    const char* Mode;
    FILE* Handle;

    File(std::string path, const char* mode)
```

```
        : Path{ path }
        , Mode{ mode }
        , Handle(fopen(path.c_str(), mode))
    {
    }

    File(const File& other)
        : Path{ other.Path }
        , Mode{ other.Mode }
        , Handle(fopen(other.Path.c_str(), other.Mode))
    {
    }

    File(File&& other) noexcept
        : Path{ std::move(other.Path) }
        , Mode{ other.Mode }
        , Handle(other.Handle)
    {
        other.Handle = nullptr;
    }

    virtual ~File()
    {
        if (Handle)
        {
            fclose(Handle);
        }
    }
```

```
    File& operator=(const File& other)
    {
        Path = other.Path;
        Mode = other.Mode;
        Handle = fopen(other.Path.c_str(), other.Mode);
        return *this;
    }

    File& operator==(File&& other)
    {
        Path = std::move(other.Path);
        Mode = other.Mode;
        Handle = other.Handle;
        other.Handle = nullptr;
        return *this;
    }
};
```

Avoid raw loops

Hand-implemented algorithms are error-prone and difficult to read. Many common algorithms, and even parallelized versions, are implemented in the algorithms library and ranges library for us and can be used with a broad number of types. Readers of such code encounter just a named algorithm which they're likely already familiar with rather than needing to interpret that from a possibly-complex loop.

```
// Avoid
struct Player
{
    const char* Name;
    int NumPoints;
};
void Avoid(const std::vector<Player>& players)
{
    using It =
std::reverse_iterator<std::vector<Player>::const_iterator
>;
    for (It it = players.rbegin(); it != players.rend();
++it)
    {
        const Player& player = *it;
        if (player.NumPoints > 25)
        {
            Player copy = player;
            copy.NumPoints--;
```

```
            DebugLog(copy.Name, copy.NumPoints);
        }
    }
}

// Encourage
struct Player
{
    const char* Name;
    int NumPoints;
};
void Encourage(const std::vector<Player>& players)
{
    using namespace std::ranges::views;

    auto result =
        players
        | filter([](Player p) { return p.NumPoints > 25;
})
        | transform([](Player p) { p.NumPoints--; return
p; })
        | reverse;
    for (const Player& p : result)
    {
        DebugLog(p.Name, p.NumPoints);
    }
}
```

Add restrictions

When using objects in a read-only way, make them `const`. When code can usefully run at compile time, make it `constexpr`. When a function can't throw any exceptions, make it `noexcept`. When fields don't need to be used outside of a class, make them `protected` or `private`. When derivation or overriding are undesirable, make classes and member functions `final`. All of these restrictions will add compiler-enforced rules that prevent misuse such as field access or enable new uses such as compile-time code execution.

```
// Avoid: requires writable strings, can't be used at
compile time, might throw
int32_t GetTotalCharacters(std::string& first,
std::string& last)
{
    return first.size() + last.size();
}

// Encourage
constexpr int32_t GetTotalCharacters(
    const std::string& first, const std::string& last)
noexcept
{
    return first.size() + last.size();
}
```

Use braced initialization

Braced initialization (`x{}` or `x = {}`) is always clearly initialization. Other forms of initialization such as with parentheses (`x()`) or nothing (`x`) are much more ambiguous. They can be mistaken for declarations, function calls, and function-style casts. Prefer braced initialization to ensure that initialization occurs:

```
// Avoid
template <typename T>
T GetDefault()
{
    T t; // Default constructor for classes but nothing
for primitives
    return t;
}
struct Point2
{
    float X{ 0 };
    float Y{ 0 };
};
std::ostream& operator<<(std::ostream& s, const Point2&
p)
{
    s << p.X << ", " << p.Y;
    return s;
}
DebugLog(GetDefault<Point2>()); // 0, 0
```

```
DebugLog(GetDefault<int>()); // undefined behavior!

// Encourage
template <typename T>
T GetDefault()
{
    T t{}; // Default constructor for classes, primitives
are value-initialized
    return t;
}
DebugLog(GetDefault<Point2>()); // 0, 0
DebugLog(GetDefault<int>()); // 0
```

Standardize error-handling

There are two main choices for error-handling in C++: exceptions and error codes. C's `errno` isn't considered a valid choice due to its reliance on global state which is not part of the call signature and not thread-safe. Codebases should choose one approach or the other to handle errors consistently and safely. For example, introducing exceptions into a codebase that uses error codes is likely to cause uncaught exceptions that crash the program.

If exceptions are chosen, they should be thrown by value and caught by `const` reference:

```
// Avoid
try
{
    throw new std::runtime_error{ "Boom!" };
}
catch (std::runtime_error* err)
{
    DebugLog(err->what()); // Boom!
    // ... memory leak here ...
}

// Encourage
try
{
    throw std::runtime_error{ "Boom!" };
}
catch (const std::runtime_error& err)
```

```
{
    DebugLog(err.what()); // Boom!
}
```

If error codes are chosen, a wrapper type such as `std::optional` is encouraged over the use of null pointers to clearly indicate to callers that the operation may not succeed. The use of `[[nodiscard]]` is also often warranted to ensure errors are handled.

```
////////
// Avoid
////////

// Caller doesn't know what happens upon error
// Caller can ignore error return values
FILE* OpenFile(const char* path, const char* mode)
{
    return fopen(path, mode);
}

FILE* handle = OpenFile("/path/to/file", "rw");
fprintf(handle, "Hello!"); // Crash if null is returned
fclose(handle);

////////////
// Encourage
////////////
```

```
// Caller clearly knows this can fail due to the
std::optional return value
// Caller can't ignore it due to the [[nodiscard]]
attribute
[[nodiscard]] std::optional<FILE*> OpenFile(const char*
path, const char* mode)
{
    FILE* handle = fopen(path, mode);
    if (!handle)
    {
        return {};
    }
    return handle;
}

// Handling the return value is required by [[nodiscard]]
std::optional<FILE*> result = OpenFile("/path/to/file",
"rw");
if (!result.has_value())
{
    DebugLog("Failed to open file");
    return;
}

// Can't directly use the result. Forced to deal with it
being optional.
// Fewer chances to dereference null and crash
FILE* handle = result.value();
```

```
fprintf(handle, "Hello!");
fclose(handle);
```

Mark overridden member functions with `override`

A `virtual` member function that overrides a base class' member function doesn't *have* to be marked that way, but it's helpful to indicate this. It provides a keyword that readers of the code can look for to know how the function fits into the class design. It also provides the compiler with a way to enforce that the function really overrides a base class version. If the function signatures subtlely don't match or the base class no longer has such a function, the compiler will catch the mistake instead of creating a new function.

```
// Avoid
struct Animal
{
    virtual void Speak(const char* message, bool
loud=false)
    {
        // By default, animals can't speak
    }
};
struct Dog : Animal
{
    // Missing "loud" parameter creates a new function
    virtual void Speak(const char* message)
    {
        DebugLog("woof: ", message);
    }
};
std::unique_ptr<Animal> a = std::make_unique<Dog>();
```

```
a->Speak("go for a walk?"); // Prints nothing because Dog
doesn't override

// Encourage
// Avoid
struct Animal
{
    virtual void Speak(const char* message, bool
loud=false)
    {
        // By default, animals can't speak
    }
};
struct Dog : Animal
{
    // Missing "loud" parameter is a compiler error
    virtual void Speak(const char* message) override
    {
        DebugLog("woof: ", message);
    }
};
std::unique_ptr<Animal> a = std::make_unique<Dog>();
a->Speak("go for a walk?"); // Never executed due to
compiler error
```

Use `using`, not `typedef`

C's `typedef` alias is still supported, but `using` is a strictly better version of it. The alias and the target are put into the familiar assignment form where the left hand side is assigned to from the right hand side. It also supports being templated, so it fits in better with generic programming.

```
// Avoid
typedef float f32;
f32 pi = 3.14f;

// Encourage
using f32 = float;
f32 pi = 3.14f;
```

```
// Avoid
#define VEC(T) std::vector<T>
VEC(float) floats;

// Encourage
template <typename T> using Vec = std::vector<T>;
Vec<float> floats;
```

Minimize function definitions in header files

Header files are typically compiled many times as many translation units directly or indirectly `#include` them. Any changes to the header file will require recompiling all the translation units that `#include` it. The linker will eventually de-duplicate these, but the compilation is slow and so build and iteration times suffer. To reduce the time it takes to compile header files, reduce the number of function definitions in them. Instead, declare functions in them and define them in translation units whenever possible.

```
////////
// Avoid
////////

// math.h
#pragma once
bool IsNearlyZero(float x)
{
    return std::abs(x) < 0.0001f;
}

////////////
// Encourage
////////////

// math.h
#pragma once
bool IsNearlyZero(float x);
```

```cpp
// math.cpp
#include "math.h"
bool IsNearlyZero(float x)
{
    return std::abs(x) < 0.0001f;
}
```

Use internal linkage for file-specific definitions

By default, entities like variables, functions, and classes have external linkage at file scope. This slows down compilation and the linker because they need to consider the possibility that some other translation unit might want to reference those entities. To speed it up, use `static` or an unnamed namespace to give those entities internal linkage and remove their candidacy for reference by other translation units.

```
// Avoid
float PI = 3.14f;

// Encourage
static float PI = 3.14f;

// Encourage
namespace
{
    float PI = 3.14f;
}
```

Use operator overloading and user-defined literals very sparingly

Overloaded operators don't really get a name and user-defined literals usually only have a terse one. As such, it's often hard for readers to understand what they're doing. Even worse, overloaded operators may appear to have one meaning while the implementation of the overloaded operator does something else. These should generally be avoided except in cases where the meaning is already well-understood. For example, the `+` operator on two `std::string` objects is clearly concatenation of the left hand operand followed by the right hand operand but the `+` operator on two `Player` objects is quite a puzzle.

```
// Avoid
struct Player
{
    int32_t Points;

    Player operator+(const Player& other)
    {
        return { Points + other.Points };
    }
};
Player a{ 100 };
Player b{ 200 };
Player c = a + b; // No conventional meaning for what + does
DebugLog(c.Points); // 300
```

```
// Encourage
struct Vector2
{
    float X;
    float Y;

    Vector2 operator+(const Vector2& other)
    {
        return { X + other.X, Y + other.Y };
    }
};
Vector2 a{ 100, 200 };
Vector2 b{ 300, 400 };
Vector2 c = a + b; // Well-understood mathematical
operator
DebugLog(c.X, c.Y); // 400, 600
```

Prefer pre-increment to post-increment

Whether we use the pre-increment operator (`++x`) or the post-increment operator (`x++`) on a primitive type like `int` makes no difference. With classes that have overloaded this operator, especially in the case of iterators, the pre-increment operator can be implemented more efficiently by removing the need to temporarily have two copies. It's generally preferable to use the pre-increment operator for this reason:

```
// Avoid: potentially slower than pre-increment
for (auto it = floats.begin(); it != floats.end(); it++)
{
    DebugLog(*it);
}

// Encourage: always the fastest way to increment
for (auto it = floats.begin(); it != floats.end(); ++it)
{
    DebugLog(*it);
}
```

Avoid template metaprogramming

The vast majority of code written should steer far clear from advanced features such as template metaprogramming. This umbrella term refers to using templates as the Turing-complete language they are to generate very complex code at compile time. Techniques such as SFINAE, not even covered in this book, should generally be the province of a few expert-level library creators such as those implementing the C++ Standard Library, testing frameworks, serialization libraries, and so forth. Almost all "normal" code should stick to "normal" features.

```
// Avoid: SFINAE like this and other TMP tricks
template<
    typename T,
    std::enable_if_t<std::is_integral<T>::value, bool> =
true>
void PrintKindOfPrimitive(T)
{
    DebugLog("it's an integer");
}
template<
    typename T,
    std::enable_if_t<std::is_floating_point<T>::value,
bool> = true>
void PrintKindOfPrimitive(T)
{
    DebugLog("it's a float");
}
```

```
PrintKindOfPrimitive(123); // it's an integer
PrintKindOfPrimitive(3.14); // it's a float

// Encourage using libraries that implement this or
avoiding the need at all
```

Use `auto` for at least long type names

Some codebases prefer the AAA style: “almost always `auto`.” This means the type of most variables is `auto` and only in a few cases is an explicit type named. Advantages include terseness and the potential avoidance of type conversion, especially when types are changed in existing code. Other codebases prefer to explicitly name all types. Advantages include readability without the need for IDE tooltips that reveal deduced types, such as when looking at code in a web browser.

Regardless of the decision, and both are popular, both camps tend to agree that very long types are hard to read and often made more clear by the use of `auto`:

```
// Avoid: type name is so long that an alias is required
to fit it on one line
using Map =
std::unordered_map<std::basic_string<char8_t>,
Game::Player*>;
using It = Map::const_iterator;
for (It it = players.begin(); it != players.end(); ++it)
{
    DebugLog(it->first, "has", it->second->Points,
"points");
}

// Encourage: use auto for at least long types like these
for (auto it = players.begin(); it != players.end();
++it)
```

```
{
    DebugLog(it->first, "has", it->second->Points,
"points");
}
```

Use compile-time polymorphism more often

Both compile-time polymorphism with templates and run-time polymorphism with `virtual` functions have valid use cases. However, most languages have limited support for compile-time polymorphism. As such, we can often overlook possibilities for performance improvements by shifting run-time work to compile-time. A lot of code that's knowable at compile-time must be determined at run-time in languages like C# while the opportunity is there in C++ to make the determination at compile-time. Idiomatic C++ tends to prefer these compile-time solutions to improve run-time performance:

```
// Avoid: run-time polymorphism when compile-time is
suitable
struct Weapon
{
    virtual void DoDamage(Player& player) = 0;
};
struct FoamDart : Weapon
{
    virtual void DoDamage(Player& player) override
    {
        player.Health--;
    }
};
struct Bazooka : Weapon
{
    virtual void DoDamage(Player& player) override
    {
```

```
        player.Health = 0;
    }
};
FoamDart w;
w.DoDamage(p); // Run-time decision

// Encourage: compile-time polymorphism when suitable
struct FoamDart
{
    void DoDamage(Player& player)
    {
        player.Health--;
    }
};
struct Bazooka
{
    void DoDamage(Player& player)
    {
        player.Health = 0;
    }
};
template <typename TWeapon>
void DoDamage(const TWeapon& weapon, Player& player)
{
    weapon.DoDamage(player);
}
FoamDart w;
DoDamage(w, player); // Compile-time decision
```

Conclusion

None of the above is gospel. There are exceptional cases for all of these guidelines. Nearly all guides will make *some* different choices than the above. Despite the conflicting guidance, all of the above are common advice in *many* guides, teams, codebases, or organizations. Still, a lot is surely missing. It's not feasible to list every best practice or idiom for C++ any more than it's feasible for a much smaller, newer language like C#.

In the end, we're not working on some abstract code. We'll work on particular codebases and it's important to be aware of the norms of those particular environments. Each will have their own written or unwritten rules, not to mention very subjective thoughts on style such as the placement of curly braces and whether indentation should be done with tabs or spaces. These certainly aren't C++-specific issues, but in the case of C++ it's wide use, large size, and long history somewhat increase the challenge.

53. Conclusion

Language

C++ and C# have quite different design goals. C++ aims to be able to be implemented by a compiler so efficiently that a programmer would never need to use another language, like C, to improve performance. In practice, assembly is sometimes used when ultimate performance is required. It's debatable as to whether this counts as another language. C++ then tries to provide as much programmer convenience as it can while also keeping to a high degree of backward-compatibility.

The goal of C# is different. It attempts to provide a lot more programmer convenience than C++ and is willing to sacrifice performance to achieve that. From the perspective of languages like Java, Python, and JavaScript, C# is much closer to the performance end of the spectrum. C# finds a middle ground. Its inclusion of structs is just one example of C#'s willingness to increase the complexity programmers need to deal with so that they can improve performance. Java is simpler because it just has classes so there's only one kind of thing that groups together variables and functions, not two.

Because C# doesn't aim for extreme performance, C# programmers aiming to achieve extreme performance often do resort to calls into other languages. Chief among them are C++ and C. This bifurcated experience itself increases the complexity of the programming environment as marshaling between the languages is required and few concepts, such as types, are shared.

Likewise, many C# features can't be used when high performance is required. Classes and arrays, for example, necessarily entail memory management and garbage collection (GC) which are nigh impossible to optimize for high performance use cases such as VR

games. Even Unity's Burst compiler is forced to put a ban on language features like these. Many Unity developers have long ago banned or minimized their use as well. The resulting programming experience, replete with cumbersome and error-prone requirements such as object pools, is far from ideal.

The same kind of criticisms are made of C++, but in the opposite direction. It's focus on performance results in many sharp edges. Variables aren't initialized by default and it's pretty easy to use a "dangling" pointer. There's a lot of "undefined behavior," too. Most of this is necessary because providing these guarantees is deemed to be too limiting or would entail overhead such as the addition of a GC.

In the end, both languages have different goals and have made decades of design choices in line with achieving those goals. Each language becomes rather unpleasant to use outside its intended purpose. C++ is a probably a poor choice for a web service and C# is probably a poor choice for training a neural network. Heroic efforts have been made to improve C++'s programmer-friendliness and C#'s performance, but these remain uphill battles even after many years of struggle.

Standard Library

C++'s standard library is much more conservative than C#'s. It's company- and industry-agnostic and sticks to well-standardized techniques and algorithms. C#'s standard library is has a lot of company-specific features, especially when it comes to Microsoft-owned technologies such as Windows. In general, it's a lot larger than the C++ Standard Library as it contains all of this company-specific functionality but also a lot of support for widely-used standards such as JSON and AES. One consequence of this broader support is that support for older features such as GDI+ are carried forward as baggage in C# or dropped at the cost of backward-compatibility.

In terms of design, the two again diverge quite a bit. C++ provides powerful language features that enable it to efficiently implement "core" types like strings and tuples in the C++ Standard Library. C# prefers to build these into the language. Where C++ provides zero-overhead extension, such as through template-based compile-time polymorphism, of the types in its standard library, C# often provides little extensibility or extensibility via mechanisms such as virtual functions that entail a runtime cost. The C# standard library is typically easier to use and more consistent across codebases but with lower performance and customizability. This is an extension and implication of the two languages' design goals to their standard libraries.

Users of either standard library ultimately turn to other libraries and frameworks to complete their apps. Whether it's Unity for a game or ASP.NET for a web service, C# apps rarely rely solely on the standard library. The same goes for C++ where its users build games on Unreal Engine or computer vision on OpenCV. Both languages are very popular so there are tons of libraries available for a wide range of tasks.

Problems Writing C#

The choice of language brings with it all the choices made by the creators of that language and its standard library. In choosing C#, we're choosing a language where many of the features require the presence of a memory manager and a GC. Consider classes. There's no way to allocate the memory where they're stored without the memory manager and no way to explicitly deallocate that memory. The `new` operator implicitly tells the memory manager to allocate memory, its use is implicitly tracked, and it's implicitly deallocated for us when no longer in use. It's not just hard or awkward to take control over the lifecycle of a C# class, it's impossible.

When we need this level of control, we're outside the C#'s comfort zone and we'll face headwinds. To illustrate, let's consider two paths we could take to solve the problem. First, we can use a subset of C# that doesn't include features like classes. This is the route taken by Unity's Burst compiler and its "High Performance C#" (HPC#) language subset. It uses structs and (unsafe) pointers instead of classes in order to provide its own memory allocation and deallocation.

The main issue with this approach is that a lot of C# language and library design assumes that classes are present. When we kick them out of the language, we lose our only mechanism that supports inheritance, virtual functions, default constructors, and reference semantics. We also make almost all C# libraries unusable as they don't conform to our language subset. The result is a very constrained environment where we end up needing to call `Dispose` functions to manually manage memory and where we cut ourselves on sharp edges like the use of uninitialized objects due to the lack of default constructors or the use of objects after calling `Dispose`. Runtime safeguards can and have been added, but with runtime

overhead and feedback on programming errors delayed to runtime. Neither is necessary in idiomatic C# where classes are used.

The second path is to keep using the whole language but in a very unidiomatic way. This has been the traditional approach to C# programming in Unity. One common example is the object pool where we avoid releasing references so that the GC doesn't run and cause a frame spike:

```
public class ParticlePool
{
    private Stack<Particle> Particles;

    public Particle Get(Color color)
    {
        if (Particles.Count > 0)
        {
            Particle p = Particles.Pop();
            p.Init(color); // Need an Init function in
addition to a constructor
            return p;
        }
        return new Particle(color);
    }

    public void Release(Particle p)
    {
        Particles.Push(p);
    }
}
```

```
// At startup, establish the pool
ParticlePool pool = new ParticlePool();

// When needed, get a particle
Particle p = pool.Get(Color.Red);

// ... use p ...

// When done, put it back in the pool
pool.Release(p);

// Nothing stopping us from using particles we released.
Causes conflicts!
p.Color = Color.Green;
```

Manual approaches like these, without any support from the language, are notoriously error-prone. Usually it's a complaint against *C++* that memory must be managed manually, but it turns out to be necessary in either of C#'s high-performance paths too. Either way, we're outside of the design goals for C# and so we run into a lot of resistance in terms of performance *and* ease-of-use barriers.

Problems Writing C++

C++ is no panacea. Its problems tend to be the other way around: more code has to be built up to make the language programmer-friendly because the defaults are often downright dangerous. C++ is outside its own comfort zone *by default* and almost always requires library support to make it usable in any practical sense. Consider the same problem of memory management. C++'s default for the `new` and `delete` operators is quite error-prone:

```
// When needed, allocate and initialize a particle
Particle* p = new Particle(Color::Red);

// ... use p ...

// When done, delete it
delete p;

// Nothing stopping us from double-deleting. Causes
crashes!
delete p;

// Nothing stopping us from using particles we deleted.
Causes crashes!
p->Color = Color::Green;
```

It's to the point that best practices discourage using these language features outside of specialty code such as classes that own the

memory through their lifecycle functions. We need to instead use library code that makes the raw language easier to use:

```
// Need a library
#include <memory>

void Foo()
{
    // When needed, allocate and initialize a particle
    std::unique_ptr<Particle> p =
std::make_unique<Particle>(Color::Red);

    // ... use p ...
} // unique_ptr's destructor deletes the Particle
```

The addition of this library code brings us to roughly the level of convenience as in idiomatic C#, but layers of libraries have overhead in terms of complexity, compile times, and verbosity. Because we opt-in to this library code, it's also easy to accidentally ignore it and use raw language features. Libraries can't save us from these mistakes. It's common to add static and dynamic analyzer tools, but none are as robust as language-level safeguards.

This is a reflection of C++'s bottom-up design. The language is extremely powerful but also extremely hard to use. The C++ Standard Library is layered on top to make it easier to use, but only for very general tasks. Additional libraries are then layered on top of these to make domain-specific tasks easier. C++ achieves great flexibility and great performance because libraries can be ignored but the language cannot. C# builds a lot into the language and thus has much less flexibility as so much is unavailable for us to opt-out of. On the other hand, this makes C# code a lot more standardized.

For example, we never see an alternative implementation of the `string` type but C++ has many: std::string, FString, QString, fbstring, CsString, CString, …

Conclusion

There is no best language, even within the domain of game programming. C# is the language of Unity, but that choice is a mixed bag of problems and benefits. C++ is the language of nearly every other game engine, but that too has many problems and benefits.

Practically, our best option is to learn the strong and weak suits of the two languages and use them for the purposes they're best suited to. A deep knowledge of each language, their standard libraries, and the surrounding world of libraries and frameworks is extremely helpful when it comes to knowing what's possible, what's feasible, which language to choose for which task, and, ultimately, how to go about the process of actually implementing in the chosen language.

Hopefully this book has delivered on *its* goal of broadening your skills so you can effectively write code for other engines, or even write C++ scripts for Unity!