Learn C++

Evyal

Contents

Chapter 0

Programming

0.0 Ubuntu

0.1 Programs and programming languages

A computer program is a sequence of instructions that directs a computer to perform certain actions in a specified order. Computer programs are typically written in a programming language, which is a language designed to facilitate the writing of instructions for computers. There are many different programming languages available, each of which caters to a different set of needs. The act (and art) of writing a program is called programming. We'll talk more specifically about how to create programs in C++ in upcoming lessons in this chapter.
When a computer is performing the actions described by the instructions in a computer program, we say it is running or executing the program. A computer will not begin execution of a program until told to do so. That typically requires the user to launch (or run or execute) the program, although programs may also be launched by other programs.

Hardware

Programs are executed on the computer's **hardware**, which consists of the physical components that make up a computer. Notable hardware found on a typical computing device includes:

- A CPU (central processing unit, often called the "brain" of the computer), which actually executes the instructions.
- Memory, where computer programs are loaded prior to execution.
- Interactive devices (e.g. a monitor, touch screen, keyboard, or mouse), which allow a person to interact with a computer.
- Storage devices (e.g. a hard drive, SSD, or flash memory), which retain information (including installed programs) even when the computer is turned off.

Software

In contrast, the term **software** broadly refers to the programs on a system that are designed to be executed on hardware.

Platforms and portability

In modern computing, programs often interact with more than just hardware – they also interact with other software on the system (particularly the operating system). The term platform refers to a compatible set of hardware and software (OS, browser, etc...) that provides an environment for software to run. For example, the term "PC" is used colloquially to mean the platform consisting of a Windows OS running on an x86-family CPU.
Platforms often provide useful services for the programs running on them. For example, a desktop application might request the operating system give them a chunk of free memory, create a file over there, or play a sound. The running program doesn't have to know how this is actually facilitated. If a program uses capabilities or services provided by the platform, it becomes dependent on that platform, and cannot be run on other

platforms without modification. A program that can be easily transferred from one platform to another is said to be portable. The act of modifying a program so that it runs on a different platform is called porting.

0.1.1 Machine Language

A computer's CPU is incapable of understanding C++. Instead, CPUs are only capable of processing instructions written in machine language (or machine code). The set of all possible machine language instructions that a given CPU can understand is called an instruction set.
Here is a sample machine language instruction: 10110000 01100001.
How these instructions are organized and interpreted is beyond the scope of this tutorial series, but it is worth noting a few things.
First, each instruction is composed of a sequence of 1s and 0s. Each individual 0 or 1 is called a binary digit, or bit for short. The number of bits in a machine language instruction varies – for example, some CPUs process instructions that are always 32 bits long, whereas some other CPUs (such as those from the x86 family, which you may be using) have instructions that can be a variable length.
Second, each family of compatible CPUs (e.g. x86, Arm64) has its own machine language, and this machine language is not compatible with the machine language of other CPU families. This means machine language programs written for one CPU family cannot be run on CPUs from a different family!

0.1.2 Assembly Languages

Machine language instructions (like 10110000 01100001) are ideal for a CPU, but are difficult for humans to understand. Since programs (at least historically) have been written and maintained by humans, it makes sense that programming languages should be designed with human needs in mind.
An assembly language (often called assembly for short) is a programming language that essentially functions as a more human-readable machine language. Here is the same instruction as above in x86 assembly language: mov al, 0x61.
Since CPUs do not understand assembly language, assembly programs must be translated into machine language before they can be executed. This translation is done by a program called an assembler. Because each assembly language instruction is typically designed to mirror an equivalent machine language instruction, the translation process is typically straightforward.
Just like each CPU family has its own machine language, each CPU family also has its own assembly language (which is designed to be assembled into machine language for that same CPU family). This means there are many different assembly languages. Although conceptually similar, different assembly languages support different instructions, use different naming conventions, etc. . .

0.1.3 Low-level languages

Machine languages and assembly languages are considered low-level languages, as these languages provide minimal abstraction from the architecture of the machine. In other words, the programming language itself is tailored to the specific instruction set architecture it will be run on.
Low-level languages have a number of notable downsides:

- Programs written in a low-level language are not portable. Since a low-level language is tailored to a specific instruction set architecture, the programs written in the language are too. Porting such programs to other architectures is typically non-trivial.
- Writing a program in a low-level language requires detailed knowledge of the architecture itself. For instance, the instruction mov al, 061h requires knowing that al refers to a CPU register available on this specific platform, and understanding how to work with that register. On a different architecture, this register might be named something different, have different limitations, or not exist at all.
- Low-level programs are hard to understand. While individual assembly instructions can be quite understandable, it can still be hard to deduce what a section of assembly code is actually doing. And since assembly programs require many instructions to do even simple tasks, they tend to be quite long
- It is hard to write assembly programs of significant complexity because the language only provides primitive capabilities. The programmer is left to implement everything they need themselves.

0.1.4 High-level Languages

To address many of the above downsides, new "high-level" programming languages such as C, C++, Pascal (and later, languages such as Java, Javascript, and Perl) were developed.
Much like assembly programs (which must be assembled to machine language), programs written in a high-level language must be translated into machine language before they can be run. There are two primary ways this is done: compiling and interpreting.

Compiled language

C++ programs are usually compiled. A compiler is a program (or collection of programs) that reads the source code of one language (usually a high-level language) and translates it into another language (usually a low-level language). For example, a C++ compiler translates C++ source code into machine code.
The machine code output by the compiler can then be packaged into an executable file (containing machine language instructions) that can distributed to others and launched by the operating system. Notably, running the executable file does not require the compiler to be installed.

Interpreted language

Alternatively, an interpreter is a program that directly executes the instructions in the source code without requiring them to be compiled first. Interpreters tend to be more flexible than compilers, but are less efficient when running programs because the interpreting process needs to be done every time the program is run. This also means the interpreter must be installed on every machine where an interpreted program will be run.
Most high-level languages can be either compiled or interpreted. Traditionally, high-level languages like C, C++, and Pascal are compiled, whereas "scripting" languages like Perl and Javascript tend to be interpreted. Some languages, like Java, use a mix of the two. We'll explore C++ compilers in more detail shortly.

The benefits of high-level languages

High-level languages are named as such because they provide a high level of abstraction from the underlying architecture.

- High-level languages allow programmers to write programs without knowing much about the platform it will be run on. This not only makes programs easier to write, it also makes them significantly more portable. If we're careful, we can write a single C++ that will compile on every platform that has a C++ compiler! A program that is designed to run on multiple platforms is said to be cross-platform.
- Programs written in a high-level language are easier to read, write, and learn because their instructions more closely resemble the natural language and mathematics that we use every day. In many cases, high-level languages require fewer instructions to perform the same tasks as low-level languages. For example, in C++ you can write a = b * 2 + 5; in one line. In assembly language, this would take 4 to 6 different instructions. This makes programs written using high-level languages more concise, which makes them easier to understand.
- High-level languages typically include additional capabilities that make it easier to perform common programming tasks, such as requesting a block of memory or manipulating text. For example, it only takes a single instruction to determine whether the characters "abc" exist within a large block of text (and if so, how many characters has to be examined until "abc" was found). This can dramatically reduce complexity and development times.

0.2 C++

0.2.1 Q: What is C++ good at?

C++ excels in situations where high performance and precise control over memory and other resources is needed. Here are a few types of applications that C++ would excel in:

- Video games
- Real-time systems (e.g. for transportation, manufacturing, etc...)
- High-performance financial applications (e.g. high frequency trading)
- Graphical applications and simulations

- Productivity / office applications
- Embedded software
- Audio and video processing
- Artificial intelligence and neural networks
- C++ also has a large number of high-quality 3rd party libraries available, which can shorten development times significantly.

0.2.2 Compiler

In order to compile C++ source code files, we use a C++ compiler. The C++ compiler sequentially goes through each source code (.cpp) file in your program and does two important tasks:
First, the compiler checks your C++ code to make sure it follows the rules of the C++ language. If it does not, the compiler will give you an error (and the corresponding line number) to help pinpoint what needs fixing. The compilation process will also be aborted until the error is fixed.
Second, the compiler translates your C++ code into machine language instructions. These instructions are stored in an intermediate file called an **object file**. The object file also contains other data that is required or useful in subsequent steps (including data needed by the linker in step 5, and for debugging in step 7).
Object files are typically named name.o or name.obj, where name is the same name as the .cpp file it was produced from.
For example, if your program had 3 .cpp files, the compiler would generate 3 object files.

0.2.3 Linker

After the compiler has successfully finished, another program called the **linker** kicks in. The linker's job is to combine all of the object files and produce the desired output file (such as an executable file that you can run). This process is called **linking**. If any step in the linking process fails, the linker will generate an error message describing the issue and then abort.
First, the linker reads in each of the object files generated by the compiler and makes sure they are valid.
Second, the linker ensures all cross-file dependencies are resolved properly. For example, if you define something in one .cpp file, and then use it in a different .cpp file, the linker connects the two together. If the linker is unable to connect a reference to something with its definition, you'll get a linker error, and the linking process will abort.
Third, the linker typically links in one or more **library files**, which are collections of precompiled code that have been "packaged up" for reuse in other programs.
Finally, the linker outputs the desired output file. Typically this will be an executable file that can be launched (but it could be a library file if that's how you've set up your project).

0.2.4 Libraries

The standard library

C++ comes with an extensive library called the **C++ Standard Library** (usually called "the standard library") that provides a set of useful capabilities for use in your programs. One of the most commonly used parts of the C++ standard library is the Input/Output library (often called "iostream"), which contains functionality for printing text on a monitor and getting keyboard input from a user.
Almost every C++ program written utilizes the standard library in some way, so it's extremely common to have the C++ standard library linked into your programs. Most C++ linkers are configured to link in the standard library by default, so this generally isn't something you need to worry about.

3rd party libraries

You can optionally link third party libraries, which are libraries that are created and distributed by independent entities (rather than as part of the C++ standard). For example, let's say you wanted to write a program that played sounds. The C++ standard library contains no such functionality. While you could write your own code to read in the sound files from disk, check to ensure they were valid, or figure out how to route the sound data to the operating system or hardware to play through the speaker – that would be a lot of work! Instead, you'd be more likely to find some existing software project that has a library that already implements all of these things for you.
In C++, libraries can be either **static** or **shared (dynamic)**. The key difference lies in how and when the library is linked with your application.

Static Libraries

Static Libraries (.a / .lib)

Linked at compile time. The code from the static library is copied into the final executable.
Pros:

- No external dependencies at runtime
- Easier distribution (just one binary)
- Good for embedded or standalone apps

Cons:

- Larger executable size
- Requires recompiling the app to update the library

Shared Libraries

Shared Libraries (.so / .dll)

Linked at runtime. The final binary is smaller, but the library must be present when the program runs.
Pros:

- Smaller executables
- Multiple programs can share the same library in memory
- Library can be updated without recompiling

Cons:

- Requires the library to be present at runtime
- Slightly more complex deployment

Comparisons

Feature	**Static Library**	**Shared Library**
File extension	`.a`	`.so`
Link time	Compile time	Runtime
Executable size	Larger	Smaller
External dependency	No	Yes
Updatable without recompiling	No	Yes
Used by multiple programs	No	Yes (shared in memory)

Table 1: Comparison of Static and Shared Libraries

0.2.5 Building

Because there are multiple steps involved, the term **building** is often used to refer to the full process of converting source code files into an executable that can be run. A specific executable produced as the result of building is sometimes called a **build**.
For complex projects, build automation tools (such as **make** or **build2**) are often used to help automate the process of building programs and running automated tests.

0.2.6 Cmake

0.2.7 Testing and Debugging

You are now able to run your executable and see what it does!
Once you can run your program, then you can test it. **Testing** is the process of assessing whether your software is working as expected. Basic testing typically involves trying different input combinations to ensure the software behaves correctly in different cases.
If the program does not behave as expected, then you will have to do some **debugging**, which is the process of finding and fixing programming errors.
We will discuss how to test and debug your programs in more detail in future chapters.

0.3 Integrated development environments (IDEs)

An Integrated Development Environment (IDE) is a piece of software designed to make it easy to develop, build, and debug your programs.
A typical modern IDE will include:

- Some way to easily load and save your code files.
- A code editor that has programming-friendly features, such as line numbering, syntax highlighting, integrated help, name completion, and automatic source code formatting.
- A basic build system that will allow you to compile and link your program into an executable, and then run it.
- An integrated debugger to make it easier to find and fix software defects.
- Some way to install plugins so you can modify the IDE or add capabilities such as version control.

Some C++ IDEs will install and configure a C++ compiler and linker for you. Others will allow you to plug in a compiler and linker of your choice (installed separately).

0.3.1 Visual Studio Code

0.4 Project

To write a C++ program inside an IDE, we typically start by creating a new project (we'll show you how to do this in a bit). A project is a container that holds all of your source code files, images, data files, etc... that are needed to produce an executable (or library, website, etc...) that you can run or use. The project also saves various IDE, compiler, and linker settings, as well as remembering where you left off, so that when you reopen the project later, the state of the IDE can be restored to wherever you left off. When you choose to compile your program, all of the .cpp files in the project will get compiled and linked.
Each project corresponds to one program. When you're ready to create a second program, you'll either need to create a new project, or overwrite the code in an existing project (if you don't want to keep it). Project files are generally IDE specific, so a project created for one IDE will need to be recreated in a different IDE.

0.4.1 Compiling your first project

If you're using g++ on the command line:

```
g++ filename.cpp -o program_name
```

Where program_name is the name you want to give to your program. This will compile and link filename.cpp.
To run it, type:

```
./program_name
```

0.4.2 Compiler Configuration

A **build configuration** (also called a build target) is a collection of project settings that determines how your IDE will build your project. The build configuration typically includes things like what the executable will be named, what directories the IDE will look in for other code and library files, whether to keep or strip out debugging information, how much to have the compiler optimize your program, etc... Generally, you will want to leave these settings at their default values unless you have a specific reason to change something.
When you create a new project in your IDE, most IDEs will set up two different build configurations for you: a release configuration, and a debug configuration.
The **debug configuration** is designed to help you debug your program, and is generally the one you will use when writing your programs. This configuration turns off all optimizations, and includes debugging information, which makes your programs larger and slower, but much easier to debug. The debug configuration is usually selected as the active configuration by default. We'll talk more about debugging techniques in a later lesson.
The **release configuration** is designed to be used when releasing your program to the public. This version is typically optimized for size and performance, and doesn't contain the extra debugging information. Because the release configuration includes all optimizations, this mode is also useful for testing the performance of your code (which we'll show you how to do later in the tutorial series).

Compiler extensions

The C++ standard defines rules about how programs should behave in specific circumstances. And in most cases, compilers will follow these rules. However, many compilers implement their own changes to the language, often to enhance compatibility with other versions of the language (e.g. C99), or for historical reasons. These compiler-specific behaviors are called compiler extensions.
Writing a program that makes use of a compiler extension allows you to write programs that are incompatible with the C++ standard. Programs using non-standard extensions generally will not compile on other compilers (that don't support those same extensions), or if they do, they may not run correctly.
Frustratingly, compiler extensions are often enabled by default. This is particularly damaging for new learners, who may think some behavior that works is part of official C++ standard, when in fact their compiler is simply over-permissive.
Because compiler extensions are never necessary, and cause your programs to be non-compliant with C++ standards, we recommend turning compiler extensions off.

Warning and error levels

When you write your programs, the compiler will check to ensure you've followed the rules of the C++ language.
In most cases, when the compiler encounters some kind of issue, it will emit diagnostic message (often called a diagnostic for short). The C++ standard does not define how diagnostic messages should be categorized, worded, or how those issues should affect the compilation of the program. However, modern compilers have conventionally adopted the following:

- A **diagnostic error** (error for short) means the compiler has decided to halt compilation, because it either cannot proceed or deems the error serious enough to stop. Diagnostic errors generated by the compiler are often called compilation errors, compiler errors, or compile errors.
- A **diagnostic warning** (warning for short) means the compiler has decided not to halt compilation. In such cases, the issue is simply ignored, and compilation proceeds.

To help you identify where the issue is, diagnostic messages typically contain both the filename and line number where the compiler found the issue, and some text about what was expected vs what was found. The actual issue may be on that line, or on a preceding line. Once you've addressed the issue causing the diagnostic, you can try compiling again to see if the associated diagnostic message is no longer generated.
In some cases, the compiler may identify code that does not violate the rules of the language, but that it believes could be incorrect. In such cases, the compiler may decide to emit a warning as a notice to the programmer that something seems amiss. Such issues can be resolved either by fixing the issue the warning is pointing out, or by rewriting the offending lines of code in such a way that the warning is no longer generated.

Best practice: Don't let warnings pile up. Resolve them as you encounter them (as if they were errors). Otherwise a warning about a serious issue may be lost amongst warnings about non-serious issues.

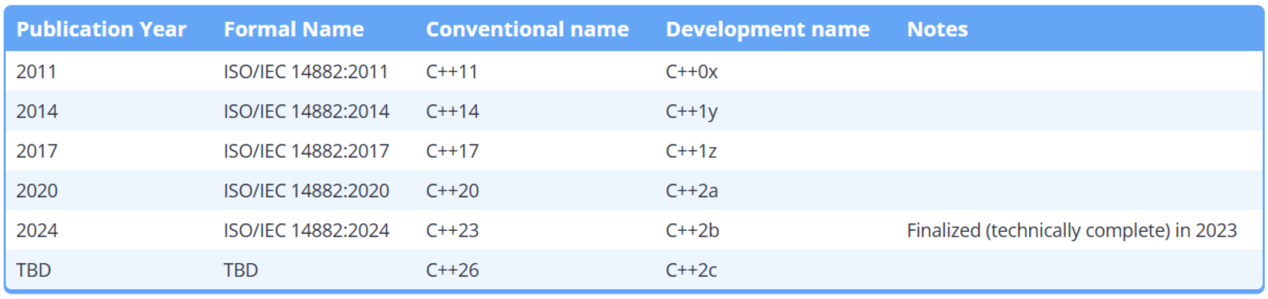
By default, most compilers will only generate warnings about the most obvious issues. However, you can request your compiler be more assertive about providing warnings, and it is generally a good idea to do so.

Best practice: Turn your warning levels up, especially while you are learning. The additional diagnostic information may help in identifying programming mistakes that can cause your program to malfunction.

It is also possible to tell your compiler to treat all warnings as if they were errors (in which case, the compiler will halt compilation if it finds any warnings). This is a good way to enforce the recommendation that you should fix all warnings (if you lack self-discipline, which most of us do).

Choosing a language standard

With many different versions of C++ available (C++98, C++03, C++11, C++14, C++17, C++20, C++23, etc...) how does your compiler know which one to use? Generally, a compiler will pick a standard to default to. Typically the default is not the most recent language standard – many default to C++14, which is missing many of the latest and greatest features.
If you wish to use a different language standard (and you probably will), you'll have to configure your IDE/-compiler to do so.
The conventional names for language standards (e.g. C++20) are based on the year the language standard was published (or expected to be published). Because the year of publication is not actually known until it is close, language standards that are early in development sometimes use a development name instead. For example, C++20 is also known as C++2a.



Publication Year	Formal Name	Conventional name	Development name	Notes
2011	ISO/IEC 14882:2011	C++11	C++0x	
2014	ISO/IEC 14882:2014	C++14	C++1y	
2017	ISO/IEC 14882:2017	C++17	C++1z	
2020	ISO/IEC 14882:2020	C++20	C++2a	
2024	ISO/IEC 14882:2024	C++23	C++2b	Finalized (technically complete) in 2023
TBD	TBD	C++26	C++2c	

Which language standard should you choose?

In professional environments, it's common to choose a language standard that is one or two versions back from the latest finalized standard (e.g. if C++20 were the latest finalized version, that means C++14 or C++17). This is typically done to ensure the compiler makers have had a chance to resolve defects, and so that best practices for new features are well understood. Where relevant, this also helps ensure better cross-platform compatibility, as compilers on some platforms may not provide full support for newer language standards immediately.

Chapter 1

C++ Basics

1.1 Structure of a program

https://www.learncpp.com/cpp-tutorial/statements-and-the-structure-of-a-program/

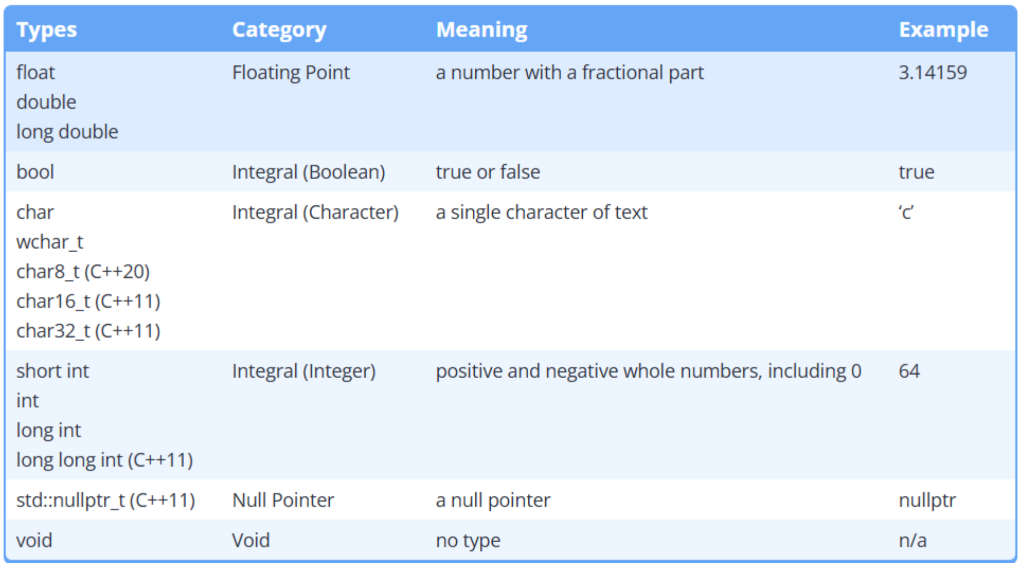
Chapter 2

Data types

Because all data on a computer is just a sequence of bits, we use a data type (often called a type for short) to tell the compiler how to interpret the contents of memory in some meaningful way.

2.1 Fundamental data types

The C++ language comes with many predefined data types available for your use. The most basic of these types are called the fundamental data types (informally sometimes called basic types or primitive types).

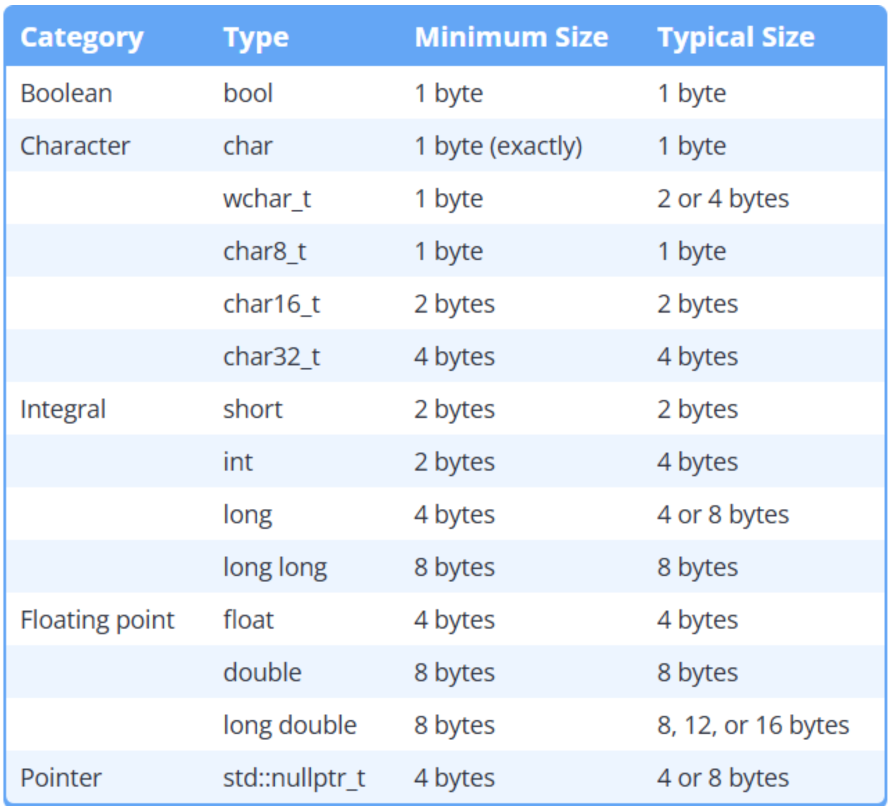


Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int int long int long long int (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

2.2 Size of data types

How much memory do objects of a given data type use? The C++ standard does not define the exact size, Instead, the standard says the following:
An object must occupy at least 1 byte (so that each object has a distinct memory address). A byte must be at least 8 bits. The integral types `char`, `short`, `int`, `long`, and `long long` have a minimum size of 8, 16, 16, 32, and 64 bits respectively. `char` and `char8_t` are exactly 1 byte (at least 8 bits).
We can have a simplified view, by making some reasonable assumptions that are generally true for modern architectures:

- A byte is 8 bits.
- Memory is byte addressable (we can access every byte of memory independently).
- We are on a 32-bit or 64-bit architecture.



Category	Type	Minimum Size	Typical Size
Boolean	bool	1 byte	1 byte
Character	char	1 byte (exactly)	1 byte
	wchar_t	1 byte	2 or 4 bytes
	char8_t	1 byte	1 byte
	char16_t	2 bytes	2 bytes
	char32_t	4 bytes	4 bytes
Integral	short	2 bytes	2 bytes
	int	2 bytes	4 bytes
	long	4 bytes	4 or 8 bytes
	long long	8 bytes	8 bytes
Floating point	float	4 bytes	4 bytes
	double	8 bytes	8 bytes
	long double	8 bytes	8, 12, or 16 bytes
Pointer	std::nullptr_t	4 bytes	4 or 8 bytes

2.2.1 The sizeof operator

In order to determine the size of data types on a particular machine, C++ provides an operator named sizeof. The sizeof operator is a unary operator that takes either a type or a variable, and returns the size of an object of that type.

```cpp
#include <iomanip> // for std::setw (which sets the width of the subsequent output)
#include <iostream>
#include <climits> // for CHAR_BIT

int main()
{
    std::cout << "A byte is " << CHAR_BIT << " bits\n\n";

    std::cout << std::left; // left justify output

    std::cout << std::setw(16) << "bool:" << sizeof(bool) << " bytes\n";
    std::cout << std::setw(16) << "char:" << sizeof(char) << " bytes\n";
    std::cout << std::setw(16) << "short:" << sizeof(short) << " bytes\n";
    std::cout << std::setw(16) << "int:" << sizeof(int) << " bytes\n";
    std::cout << std::setw(16) << "long:" << sizeof(long) << " bytes\n";
    std::cout << std::setw(16) << "long long:" << sizeof(long long) << " bytes\n";
    std::cout << std::setw(16) << "float:" << sizeof(float) << " bytes\n";
    std::cout << std::setw(16) << "double:" << sizeof(double) << " bytes\n";
    std::cout << std::setw(16) << "long double:" << sizeof(long double) << " bytes\n";

    return 0;
}
```

2.3 Type Conversions

2.3.1 Implicit type conversion

Consider the following program:

```cpp
#include <iostream>

void print(double x) // print takes a double parameter
{
  std::cout << x << '\n';
}

int main()
{
  print(5); // what happens when we pass an int value?

  return 0;
}
```

In the above example, the print() function has a parameter of type double but the caller is passing in the value 5 which is of type int. What happens in this case?
In most cases, C++ will allow us to convert values of one fundamental type to another fundamental type. The process of converting data from one type to another type is called type conversion. Thus, the int argument 5 will be converted to double value 5.0 and then copied into parameter x. The print() function will print this value.
When the compiler does type conversion on our behalf without us explicitly asking, we call this **implicit type conversion.** The above example illustrates this – nowhere do we explicitly tell the compiler to convert integer value 5 to double value 5.0. Rather, the function is expecting a double value, and we pass in an integer argument. The compiler will notice the mismatch and implicitly convert the integer to a double.

Implicit type conversion warnings

Although implicit type conversion is sufficient for most cases where type conversion is needed, there are a few cases where it is not. Consider the following program, which is similar to the example above:

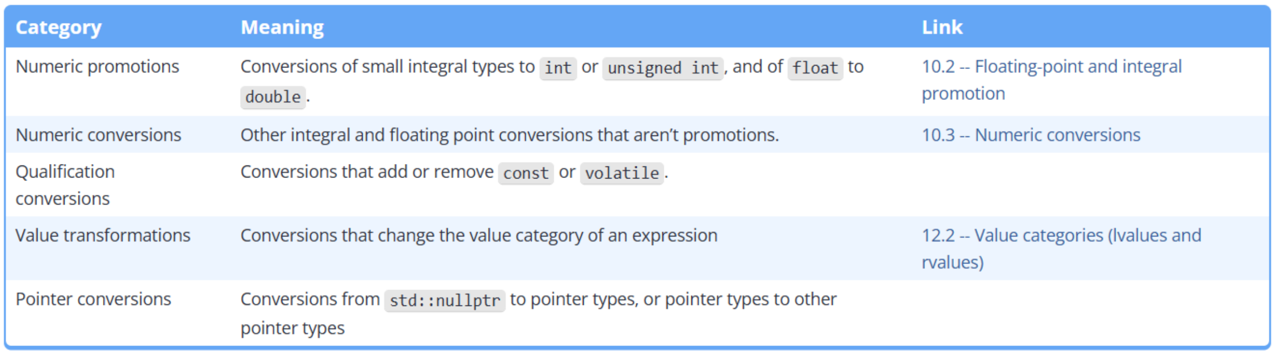
```cpp
#include <iostream>

void print(int x) // print now takes an int parameter
{
  std::cout << x << '\n';
}

int main()
{
  print(5.5); // warning: we're passing in a double value

  return 0;
}
```

In this program, we've changed print() to take an int parameter, and the function call to print() is now passing in double value 5.5. Similar to the above, the compiler will use implicit type conversion in order to convert double value 5.5 into a value of type int, so that it can be passed to function print().
Unlike the initial example, when this program is compiled, your compiler will generate some kind of a warning about a possible loss of data. And because you have "treat warnings as errors" turned on (you do, right?), your compiler will abort the compilation process.
When compiled and run, this program prints the following:

```
5
```

When implicit type conversion happens

Implicit type conversion (also called automatic type conversion or coercion) is performed automatically by the compiler when an expression of some type is supplied in a context where some other type is expected. The



Category	Meaning	Link
Numeric promotions	Conversions of small integral types to `int` or `unsigned int`, and of `float` to `double`.	10.2 -- Floating-point and integral promotion
Numeric conversions	Other integral and floating point conversions that aren't promotions.	10.3 -- Numeric conversions
Qualification conversions	Conversions that add or remove `const` or `volatile`.	
Value transformations	Conversions that change the value category of an expression	12.2 -- Value categories (lvalues and rvalues)
Pointer conversions	Conversions from `std::nullptr` to pointer types, or pointer types to other pointer types	

vast majority of type conversions in C++ are implicit type conversions. For example, implicit type conversion happens in all of the following cases:
When initializing (or assigning a value to) a variable with a value of a different data type:

```cpp
double d{ 3 }; // int value 3 implicitly converted to type double
d = 6; // int value 6 implicitly converted to type double
```

And in many other cases.

The standard conversions

As part of the core language, the C++ standard defines a collection of conversion rules known as the "standard conversions". The standard conversions specify how various fundamental types (and certain compound types, including arrays, references, pointers, and enumerations) convert to other types within that same group.
As of C++23, there are 14 different standard conversions. These can be roughly grouped into 5 general categories:
For example, converting an int value to a float value falls under the numeric conversions category, so the compiler to perform such a conversion, the compiler simply need apply the int to float numeric conversion rules.
Note: Type conversion can fail
If the compiler can't find an acceptable conversion, then the compilation will fail with a compile error. Type conversions can fail for any number of reasons. For example, the compiler might not know how to convert a value between the original type and the desired type. For example:

```cpp
int x { "14" };
```

The because there isn't a standard conversion from the string literal "14" to int, the compiler will produce an error.
In other cases, specific features may disallow some categories of conversions. For example:

```cpp
int x { 3.5 }; // brace-initialization disallows conversions that result
    in data loss
```

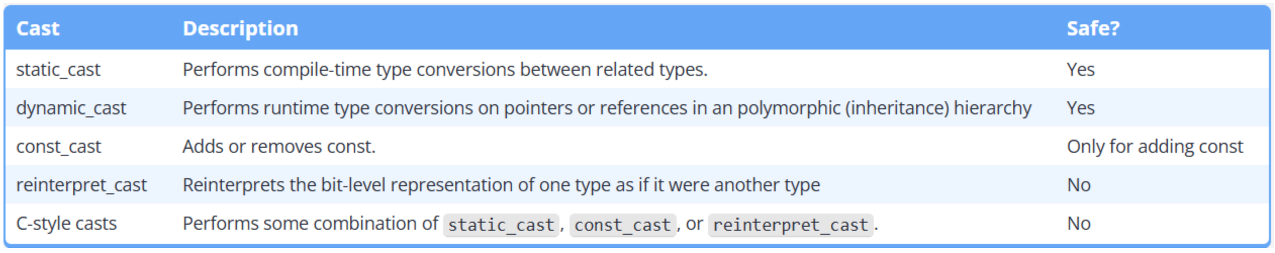
Even though the compiler knows how to convert a double value to an int value, narrowing conversions are disallowed when using brace-initialization.

2.3.2 Explicit type conversion

C++ supports a second method of type conversion, called explicit type conversion. Explicit type conversion allow us (the programmer) to explicitly tell the compiler to convert a value from one type to another type, and that we take full responsibility for the result of that conversion. If such a conversion results in the loss of value, the compiler will not warn us.
To perform an explicit type conversion, in most cases we'll use the `static_cast` operator.

2.4 Type casting

C++ comes with a number of different **type casting operators** (more commonly called **casts**) that can be used by the programmer to have the compiler perform type conversion. Because casts are explicit requests by the programmer, this form of type conversion is often called an **explicit type conversion** (as opposed to implicit type conversion, where the compiler performs a type conversion automatically).



Cast	Description	Safe?
static_cast	Performs compile-time type conversions between related types.	Yes
dynamic_cast	Performs runtime type conversions on pointers or references in an polymorphic (inheritance) hierarchy	Yes
const_cast	Adds or removes const.	Only for adding const
reinterpret_cast	Reinterprets the bit-level representation of one type as if it were another type	No
C-style casts	Performs some combination of `static_cast`, `const_cast`, or `reinterpret_cast`.	No

C++ supports 5 different types of casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, and `C-style casts`. The first four are sometimes referred to as **named casts**.
Each cast works the same way. As input, the cast takes an expression (that evaluates to a value or an object), and a target type. As output, the cast returns the result of the conversion.
Because they are the most commonly used casts, we'll cover `C-style casts` and `static_cast` in this lesson.

2.4.1 C-style cast

In standard C programming, casting is done via operator(), with the name of the type to convert to placed inside the parentheses, and the value to convert to placed immediately to the right of the closing parenthesis. In C++, this type of cast is called a **C-style cast**. You may still see these used in code that has been converted from C.

```
#include <iostream>

int main()
{
    int x { 10 };
    int y { 4 };

    std::cout << (double)x / y << '\n'; // C-style cast of x to double

    return 0;
}
```

C++ also provides an alternative form of C-style cast known as a function-style cast, which resembles a function call:

```
std::cout << double(x) / y << '\n'; //  // function-style cast of x to
    double
```

There are a couple of significant reasons that C-style casts are generally avoided in modern C++.
First, although a C-style cast appears to be a single cast, it can actually perform a variety of different conversions depending on how it is used. This can include a static cast, a const cast, or a reinterpret cast (the latter two of which we mentioned above you should avoid). A C-style cast does not make it clear which cast(s) will actual be performed, which not only makes your code that much harder to understand, but also opens the door for inadvertent misuse (where you think you're implementing a simple cast and you end up doing something dangerous instead). Often this will end up producing an error that isn't discovered until runtime.
Also, because C-style casts are just a type name, parenthesis, and variable or value, they are both difficult to identify (making your code harder to read) and even more difficult to search for.
In contrast, the named casts are easy to spot and search for, make it clear what they are doing, are limited in their abilities, and will produce a compilation error if you try to misuse them.

Best practice: Avoid using C-style casts.

2.4.2 static_cast

By far the most used cast in C++ is the **static cast** operator, which is accessed via the `static_cast` keyword.
`static_cast` is used when we want to explicitly convert a value of one type into a value of another type.
`static_cast` takes the value from an expression as input, and returns that value converted into the type specified by new_type (e.g. int, bool, char, double).
Example 1:

```cpp
#include <iostream>

int main()
{
    char c { 'a' };
    std::cout << static_cast<int>(c) << '\n'; // prints 97 rather than a

    return 0;
}
```

To perform a static cast, we start with the static_cast keyword, and then place the type to convert to inside angled brackets. Then inside parenthesis, we place the expression whose value will be converted. Note how much the syntax looks like a function call to a function named static_cast¡type¿() with the expression whose value will be converted provided as an argument! Static casting a value to another type of value returns a temporary object that has been direct-initialized with the converted value.
Example 2:

```cpp
#include <iostream>

void print(int x)
{
  std::cout << x << '\n';
}

int main()
{
  print( static_cast<int>(5.5) ); // explicitly convert double value 5.5
      to an int

  return 0;
}
```

Because we're now explicitly requesting that double value 5.5 be converted to an int value, the compiler will not generate a warning about a possible loss of data upon compilation (meaning we can leave "treat warnings as errors" enabled).
There are two important properties of static_cast.
First, static_cast provides compile-time type checking. If we try to convert a value to a type and the compiler doesn't know how to perform that conversion, we will get a compilation error.

```cpp
// a C-style string literal can't be converted to an int, so the following
    is an invalid conversion
int x { static_cast<int>("Hello") }; // invalid: will produce compilation
    error
```

Second, static_cast is (intentionally) less powerful than a C-style cast, as it will prevent certain kinds of dangerous conversions (such as those that require reinterpretation or discarding const).

Best practice: Favor `static_cast` when you need to convert a value from one type to another type.

2.5 Type aliases

In C++, **using** is a keyword that creates an alias for an existing data type. To create such a type alias, we use the using keyword, followed by a name for the type alias, followed by an equals sign and an existing data type. For example:

```cpp
using Distance = double; // define Distance as an alias for type double
```

Once defined, a type alias can be used anywhere a type is needed. For example, we can create a variable with the type alias name as the type:

```cpp
Distance milesToDestination{ 3.4 }; // defines a variable of type double
```

Note: Type aliases are not distinct types, When the compiler encounters a type alias name, it will substitute in the aliased type.

Best practice: Name your type aliases starting with a capital letter and do not use a suffix (unless you have a specific reason to do otherwise).

Because scope is a property of an identifier, type alias identifiers follow the same scoping rules as variable identifiers: a type alias defined inside a block has block scope and is usable only within that block, whereas a type alias defined in the global namespace has global scope and is usable to the end of the file.

Using type aliases for platform independent coding

One of the primary uses for type aliases is to hide platform specific details. On some platforms, an int is 2 bytes, and on others, it is 4 bytes. Thus, using int to store more than 2 bytes of information can be potentially dangerous when writing platform independent code.
Because char, short, int, and long give no indication of their size, it is fairly common for cross-platform programs to use type aliases to define aliases that include the type's size in bits. For example, int8_t would be an 8-bit signed integer, int16_t a 16-bit signed integer, and int32_t a 32-bit signed integer. Using type aliases in this manner helps prevent mistakes and makes it more clear about what kind of assumptions have been made about the size of the variable.
In order to make sure each aliased type resolves to a type of the right size, type aliases of this kind are typically used in conjunction with preprocessor directives:

```
#ifdef INT_2_BYTES
using int8_t = char;
using int16_t = int;
using int32_t = long;
#else
using int8_t = char;
using int16_t = short;
using int32_t = int;
#endif
```

Using type aliases to make complex types easier to read

Although we have only dealt with simple data types so far, in advanced C++, types can be complicated and lengthy to manually enter on your keyboard. For example, you might see a function and variable defined like this, and it's much easier to use a type alias:

```
#include <string> // for std::string
#include <vector> // for std::vector
#include <utility> // for std::pair

using VectPairSI = std::vector<std::pair<std::string, int>>; // make
    VectPairSI an alias for this crazy type

bool hasDuplicates(VectPairSI pairlist) // use VectPairSI in a function
    parameter
{
    // some code here
    return false;
}

int main()
{
     VectPairSI pairlist; // instantiate a VectPairSI variable

     return 0;
}
```

2.5.1 Typedef

A typedef (which is short for “type definition”) is an older way of creating an alias for a type. To create a typedef alias, we use the typedef keyword:

```cpp
// The following aliases are identical
typedef long Miles;
using Miles = long;
```

Best practice: Prefer type aliases over typedefs.

2.5.2 Downsides and conclusion

While type aliases offer some benefits, they also introduce yet another identifier into your code that needs to be understood. If this isn’t offset by some benefit to readability or comprehension, then the type alias is doing more harm than good.
A poorly utilized type alias can take a familiar type (such as std::string) and hide it behind a custom name that needs to be looked up. In some cases (such as with smart pointers, which we’ll cover in a future chapter), obscuring the type information can also be harmful to understanding how the type should be expected to work.
For this reason, type aliases should be used primarily in cases where there is a clear benefit to code readability or code maintenance. This is as much of an art as a science. Type aliases are most useful when they can be used in many places throughout your code, rather than in fewer places.

Best practice: Use type aliases judiciously, when they provide a clear benefit to code readability or code maintenance.

2.6 Type deduction

Type deduction (also sometimes called type inference) is a feature that allows the compiler to deduce the type of an object from the object’s initializer. When defining a variable, type deduction can be invoked by using the auto keyword can be used in place of the variable’s type:

```cpp
int main()
{
    auto d { 5.0 }; // 5.0 is a double literal, so d will be deduced as a
        double
    auto i { 1 + 2 }; // 1 + 2 evaluates to an int, so i will be deduced
        as an int
    auto x { i }; // i is an int, so x will be deduced as an int

    return 0;
}
```

Note: Type deduction must have something to deduce from

Type deduction will not work for objects that either do not have initializers or have empty initializers. It also will not work when the initializer has type void (or any other incomplete type). Thus, the following is not valid:

```cpp
#include <iostream>

void foo()
{
}

int main()
{
    auto a;             // The compiler is unable to deduce the type of a
    auto b { };         // The compiler is unable to deduce the type of b
    auto c { foo() }; // Invalid: c can't have type incomplete type void

    return 0;
}
```

In most cases, type deduction will drop the const from deduced types. For example:

```
int main()
{
    const int a { 5 }; // a has type const int
    auto b { a };      // b has type int (const dropped)

    return 0;
}
```

2.6.1 Benefits and downsides

Type deduction is not only convenient, but also has a number of other benefits.
First, if two or more variables are defined on sequential lines, the names of the variables will be lined up, helping to increase readability:

```
// harder to read
int a { 5 };
double b { 6.7 };

// easier to read
auto c { 5 };
auto d { 6.7 };
```

Second, type deduction only works on variables that have initializers, so if you are in the habit of using type deduction, it can help avoid unintentionally uninitialized variables:

```
int x; // oops, we forgot to initialize x, but the compiler may not
    complain
auto y; // the compiler will error out because it can't deduce a type for y
```

Third, you are guaranteed that there will be no unintended performance-impacting conversions:

```
std::string_view getString();   // some function that returns a
    std::string_view

std::string s1 { getString() }; // bad: expensive conversion from
    std::string_view to std::string (assuming you didn't want this)
auto s2 { getString() };        // good: no conversion required
```

Type deduction also has a few downsides.

```
auto y { 5 }; // oops, we wanted a double here but we accidentally
    provided an int literal
```

Here's another example:

```
#include <iostream>

int main()
{
     auto x { 3 };
     auto y { 2 };

     std::cout << x / y << '\n'; // oops, we wanted floating point
         division here

     return 0;
}
```

In this example, it's less clear that we're getting an integer division rather than a floating-point division.

Best practice: Use type deduction for your variables when the type of the object doesn't matter.

Favor an explicit type when you require a specific type that differs from the type of the initializer, or when your object is used in a context where making the type obvious is useful.

2.6.2 Type deduction for functions

Since the compiler already has to deduce the return type from the return statement (to ensure that the value can be converted to the function's declared return type), in C++14, the auto keyword was extended to do function return type deduction. This works by using the auto keyword in place of the function's return type.

```
auto add(int x, int y)
{
    return x + y;
}
```

When using an auto return type, all return statements within the function must return values of the same type, otherwise an error will result. For example:

```
auto someFcn(bool b)
{
    if (b)
        return 5; // return type int
    else
        return 6.7; // return type double
}
```

In the above function, the two return statements return values of different types, so the compiler will give an error.

Best practice: Prefer explicit return types over return type deduction (except in cases where the return type is unimportant, difficult to express, or fragile).

Fragile means cases where return type is likely to change if the implementation changes.

Chapter 3

Files

3.1 Programs with multiple code files

Programs with multiple code files
`https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/`
As programs get larger, it is common to split them into multiple files for organizational or reusability purposes.

Example: single-file program that wouldn't compile

Consider a multiple file program made of these two files:
add.cpp:

```cpp
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

main.cpp:

```cpp
1  #include <iostream>
2  
3  int main()
4  {
5      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
6      return 0;
7  }
8  
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

Your compiler may compile either add.cpp or main.cpp first. Either way, main.cpp will fail to compile. The reason is that when the compiler reaches line 5 of `main.cpp`, it doesn't know what identifier `add` is.

Remember, the compiler compiles each file individually. It does not know about the contents of other code files, or remember anything it has seen from previously compiled code files.

Our options for a solution here are two: either place the definition of function add before function main, or satisfy the compiler with a forward declaration. In this case, because we want to keep the function add is in another file, the reordering option isn't possible.
The solution is simply adding a forward declaration inside of `main.cpp`.

```cpp
1  #include <iostream>
2  
3  int add(int x, int y); // needed so main.cpp knows that add() is a
       function defined elsewhere
4  
5  int main()
6  {
7      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
8      return 0;
}
```

Now, when the compiler is compiling `main.cpp`, it will know what identifier add is and be satisfied. The linker will connect the function call to add in `main.cpp` to the definition of function add in `add.cpp`.

3.2 Naming collisions

C++ requires that all identifiers be non-ambiguous. If two identical identifiers are introduced into the same program in a way that the compiler or linker can't tell them apart, the compiler or linker will produce an error. This error is generally referred to as a **naming collision** (or naming conflict).

If the colliding identifiers are introduced into the same file, the result will be a compiler error. If the colliding identifiers are introduced into separate files belonging to the same program, the result will be a linker error.

To avoid this, one possible solution is to insert one of the two functions inside a `namespace`.

3.3 Namespaces

3.3.1 The global namespace

3.4 Preprocessor

When you compile your project, you might expect that the compiler compiles each code file exactly as you've written it. This actually isn't the case.

Instead, prior to compilation, each code (.cpp) file goes through a preprocessing phase. In this phase, a program called the preprocessor makes various changes to the text of the code file. The preprocessor does not actually modify the original code files in any way – rather, all changes made by the preprocessor happen either temporarily in-memory or using temporary files.

Most of what the preprocessor does is fairly uninteresting. For example, it strips out comments, and ensures each code file ends in a newline. However, the preprocessor does have one very important role: it is what processes `#include` directives (which we'll discuss more in a moment).

When the preprocessor has finished processing a code file, the result is called a translation unit. This translation unit is what is then compiled by the compiler.

3.4.1 Preprocessor directives

When the preprocessor runs, it scans through the code file (from top to bottom), looking for preprocessor directives. Preprocessor directives (often just called directives) are instructions that start with a `#` symbol and end with a newline (NOT a semicolon). These directives tell the preprocessor to perform certain text manipulation tasks. Note that the preprocessor does not understand C++ syntax – instead, the directives have their own syntax (which in some cases resembles C++ syntax, and in other cases, not so much).

3.4.2 Include

When you `#include` a file, the preprocessor replaces the `#include` directive with the contents of the included file. The included contents are then preprocessed (which may result in additional `#includes` being preprocessed recursively), then the rest of the file is preprocessed.

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
    return 0;
}
```

When the preprocessor runs on this program, the preprocessor will replace `#include` ¡iostream¿ with the contents of the file named “iostream” and then preprocess the included content and the rest of the file.

When including multiple a file in another file, some violations of the ODR (one definition rule) might occur. To circumvent this, **header guards** are used, which we will be discussing later.

3.4.3 Macro defines

The **#define** directive can be used to create a macro. In C++, a macro is a rule that defines how input text is converted into replacement output text. There are two basic types of macros: object-like macros, and function-like macros.

Function-like macros act like functions, and serve a similar purpose, but we are not interested in them. Their use is generally considered unsafe, and almost anything they can do can be done by a normal function.

Object-like macros conversely are very useful, and can be defined in one of two ways:

```cpp
#include <iostream>

#define MY_NAME "Alex"

int main()
{
    std::cout << "My name is: " << MY_NAME << '\n';

    return 0;
}
```

The preprocessor converts the above into the following:

```cpp
// The contents of iostream are inserted here

int main()
{
    std::cout << "My name is: " << "Alex" << '\n';

    return 0;
}
```

3.4.4 Conditional compilation

The conditional compilation preprocessor directives allow you to specify under what conditions something will or won't compile. There are quite a few different conditional compilation directives, but we'll only cover a few that are used the most often: **#ifdef**, **#ifndef**, and **#endif**.

```cpp
#include <iostream>

#define PRINT_JOE

int main()
{
#ifdef PRINT_JOE
    std::cout << "Joe\n"; // will be compiled since PRINT_JOE is defined
#endif

#ifdef PRINT_BOB
    std::cout << "Bob\n"; // will be excluded since PRINT_BOB is not
        defined
#endif

    return 0;
}
```

However, their usefulness comes primarily from the role they play inside of header file, which we will be disccusing next.

3.5 Header Files

In section 3.1, we discussed how programs can be split across multiple files. We also discussed how forward declarations are used to allow the code in one file to access something defined in another file.
When programs contain only a few small files, manually adding a few forward declarations to the top of each file isn't too bad. However, as programs grow larger (and make use of more files and functions), having

to manually add a large number of (possibly different) forward declarations to the top of each file becomes extremely tedious. For example, if you have a 5 file program, each of which requires 10 forward declarations, you're going to have to copy/paste in 50 forward declarations. Now consider the case where you have 100 files and they each require 100 forward declarations. This simply doesn't scale!

To address this issue, C++ programs typically take a different approach. C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs. The other type of file is called a header file. Header files usually have a .h extension, but you will occasionally see them with a .hpp extension or no extension at all.

Header Files: Conventionally, header files are used to propagate a bunch of related forward declarations into a code file.

Going back to the example we were discussing previously, containing add.cpp and main.cpp, that looked like this:
add.cpp:

```
int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

int add(int x, int y); // forward declaration using function prototype

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

Here's our completed header file:

```
// We really should have a header guard here, but will omit it for
    simplicity (we'll cover header guards next)

int add(int x, int y); // function prototype for add.h
```

Now we will have to modify our add.cpp and main.cpp to tell the preprocessor to include the forward declaration inside of them:
add.cpp

```
#include "add.h" // Insert contents of add.h at this point.  Note use of
    double quotes here.

int add(int x, int y)
{
    return x + y;
}
```

main.cpp

```
#include "add.h" // Insert contents of add.h at this point.  Note use of
    double quotes here.
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

When the preprocessor processes the `#include` "add.h" line, it copies the contents of add.h into the current file at that point. Because our add.h contains a forward declaration for function add(), that forward declaration will be copied into main.cpp. The end result is a program that is functionally the same as the one where we manually added the forward declaration at the top of main.cpp.

Consequently, our program will compile and link correctly.

Possible violation of the ODR

Including definitions in a header file results in a violation of the one-definition rule. Let's say out add.h looked like this:

```cpp
// definition for add() in header file -- don't do this!
int add(int x, int y)
{
    return x + y;
}
```

When main.cpp is compiled, the `#include` "add.h" will be replaced with the contents of add.h and then compiled. Therefore, the compiler will compile something that looks like this:

```cpp
// from add.h:
int add(int x, int y)
{
    return x + y;
}

// contents of iostream header here

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';

    return 0;
}
```

That will compile just fine. And this:

```cpp
int add(int x, int y)
{
    return x + y;
}
```

That will also compile just fine.

Finally, the linker will run. And the linker will see that there are now two definitions for function add(): one in main.cpp, and one in add.cpp. This is a violation of ODR part 2, which states, "Within a given program, a variable or normal function can only have one definition."

Using header files

Source files should include their paired header: In C++, it is a best practice for code files to `#include` their paired header file (if one exists). This allows the compiler to catch certain kinds of errors at compile time instead of link time.

Note: Do not `#include` .cpp files. Although the preprocessor will happily do so, you should generally not `#include` .cpp files. These should be added to your project and compiled.
There are number of reasons for this:

- Doing so can cause naming collisions between source files.
- In a large project it can be hard to avoid one definition rules (ODR) issues.
- Any change to such a .cpp file will cause both the .cpp file and any other .cpp file that includes it to recompile, which can take a long time. Headers tend to change less often than source files.
- It is non-conventional to do so.

Header file best practices

Here are a few more recommendations for creating and using header files.

- Always include header guards (we'll cover these next lesson).
- Do not define variables and functions in header files (for now).
- Give a header file the same name as the source file it's associated with (e.g. grades.h is paired with grades.cpp).
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
- Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files, to avoid inadvertent transitive includes.
- A header file should `#include` any other headers containing functionality it needs. Such a header should compile successfully when `#include`d into a .cpp file by itself.
- Only `#include` what you need (don't include everything just because you can).
- Do not `#include` .cpp files.
- Prefer putting documentation on what something does or how to use it in the header. It's more likely to be seen there. Documentation describing how something works should remain in the source files.

3.6 Header Guards

We already noted that a variable or function identifier can only have one definition (the one definition rule) or it will cause a compile error.

With header files, it's quite easy to end up in a situation where a definition in a header file gets included more than once. This can happen when a header file `#includes` another header file (which is common).
The good news is that we can avoid the above problem via a mechanism called a **header guard** (also called an **include guard**). Header guards are conditional compilation directives that take the following form:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

When this header is `#included`, the preprocessor will check whether `SOME_UNIQUE_NAME_HERE` has been previously defined in this translation unit. If this is the first time we're including the header, `SOME_UNIQUE_NAME_HERE` will not have been defined. Consequently, it `#defines` `SOME_UNIQUE_NAME_HERE` and includes the contents of the file. If the header is included again into the same file, `SOME_UNIQUE_NAME_HERE` will already have been defined from the first time the contents of the header were included, and the contents of the header will be ignored (thanks to the `#ifndef`).

All of your header files should have header guards on them.

Naming conventions

By convention the name of an header guard is set to the full filename of the header file, typed in all caps, using underscores for spaces or punctuation. For example in square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

Note: Header guards do not prevent a header from being included once into different code files. Therefore, even when using header guards, defining a function inside of an header file that is included in multiple files will cause a violation of the ODR.

As an example:
square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength); // forward declaration for
    getSquarePerimeter

#endif
```

square.cpp:

```
#include "square.h"  // square.h is included once here

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << " sides\n";
    std::cout << "a square of length 5 has perimeter length " <<
        getSquarePerimeter(5) << '\n';

    return 0;
}
```

This program will compile, but the linker will complain about your program having multiple definitions for identifier getSquareSides!

`#pragma` once

Modern compilers support a simpler, alternate form of header guards using the `#pragma` preprocessor directive:

```
#pragma once

// your code here
```

Chapter 4

Operators

4.1

https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/

Chapter 5

Functions

5.1 Introduction to functions

A function is a reusable sequence of statements designed to do a particular job. As programs start to get longer and longer, putting all the code inside the main() function becomes increasingly hard to manage. Functions provide a way for us to split our programs into small, modular chunks that are easier to organize, test, and use.

Most programs use many functions. The C++ standard library comes with plenty of already-written functions for you to use – however, it's just as common to write your own. Functions that you write yourself are called **user-defined functions**.

How do functions work? A program will be executing statements sequentially inside one function when it encounters a function call. A function call tells the CPU to interrupt the current function and execute another function. The CPU essentially "puts a bookmark" at the current point of execution, executes the function named in the function call, and then returns to the point it bookmarked and resumes execution.

Declaration

```
returnType functionName() // This is the function header (tells the
    compiler about the existence of the function)
{
    // This is the function body (tells the compiler what the function
        does)
}
```

The first line is informally called the **function header**. The curly braces and statements in-between are called the **function body**.

Calling a function

To call a function, we use the function's name followed by a set of parentheses (e.g. functionName() calls the function whose name is functionName). Conventionally, the parenthesis are placed adjacent to the function name (with no whitespace between them).
For now, a function must be defined before it can be called.

```
void doPrint()
{
    std::cout << "In doPrint()\n";
}

doPrint();
```

Note: Functions can call functions that call other functions

```
void doB()
{
    std::cout << "In doB()\n";
}
```

```cpp


void doA()
{
    std::cout << "Starting doA()\n";

    doB();

    std::cout << "Ending doA()\n";
}
```

Note: A function whose definition is placed inside another function is a nested function. Unlike some other programming languages, in C++, functions cannot be nested.

5.2 Function return values

When you write a user-defined function, you get to determine whether your function will return a value back to the caller or not. To return a value back to the caller, two things are needed.

First, your function has to indicate what type of value will be returned. This is done by setting the function's return type, which is the type that is defined before the function's name.

Second, inside the function that will return a value, we use a return statement to indicate the specific value being returned to the caller. The return statement consists of the return keyword, followed by an expression (sometimes called the return expression), ending with a semicolon.

```cpp
int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}
```

When the return statement is executed:

- The return expression is evaluated to produce a value.
- The value produced by the return expression is copied back to the caller. This copy is called the return value of the function.
- The function exits, and control returns to the caller.

The process of returning a copied value back to the caller is named **return by value**.

5.2.1 Revisiting main()

In C++, there are two special requirements for `main()`:

- `main()` is required to return an `int`.
- Explicit function calls to `main()` are disallowed.

Note: A value-returning function that does not return a value will produce undefined behavior.
Note: A function that returns a value is called a **value-returning function**. A function is value-returning if the return type is anything other than void.
Note: A value-returning function must return a value of that type, otherwise undefined behavior will result.
Note: A value-returning function can only return a single value back to the caller each time it is called.

5.3 Void functions

Functions are not required to return a value back to the caller. To tell the compiler that a function does not return a value, a return type of void is used. For example:

```cpp
void printHi()
{
    std::cout << "Hi" << '\n';

    // This function does not return a value so no return statement is
        needed
}
```

Void functions don't need a return statement, a void function will automatically return to the caller at the end of the function.

A return statement (with no return value) can be used in a void function – such a statement will cause the function to return to the caller at the point where the return statement is executed (note that this can be quite useful in some situations!). This is the same thing that happens at the end of the function anyway. However, putting an empty return statement at the end of a void function is redundant.

Note: Void functions can't be used in expression that require a value. For example:

```cpp
void printHi()
{
    std::cout << "Hi" << '\n';
}

int main()
{
    printHi(); // okay: function printHi() is called, no value is returned

    std::cout << printHi(); // compile error

    return 0;
}
```

Note: As mentioned before, trying to return a value from a non-value returning function will result in a compilation error:

```cpp
void printHi() // This function is non-value returning
{
    std::cout << "In printHi()" << '\n';

    return 5; // compile error: we're trying to return a value
}
```

5.4 function parameters

Let's say we want to make a function that prints a number.

```cpp
// This function won't compile
void printDouble()
{
  std::cout << num << " doubled is: " << num * 2 << '\n';
}

int main()
{
  int num { getValueFromUser() };
  printDouble();

  return 0;
}
```

The reason why this won't work is due to variables scope, which we will cover in a further chapter

To make this work we use **function parameters** (or **function arguments**).

A function parameter is a variable used in the header of a function. Function parameters work almost identically to variables defined inside the function, but with one difference: they are initialized with a value provided by the caller of the function.

Function parameters are defined in the function header by placing them in between the parenthesis after the function name, with multiple parameters being separated by commas.

```cpp
int add(int x, int y)
{
    return x + y;
}

// Function Call
add(2, 3); // 2 and 3 are the arguments passed to function add()
```

5.4.1 Pass by value

When a function is called, all of the parameters of the function are created as variables, and the value of each of the arguments is copied into the matching parameter (using copy initialization). This process is called pass by value. Function parameters that utilize pass by value are called value parameters.
For example:

```cpp
#include <iostream>

// The values of x and y are passed in by the caller
void printValues(int x, int y)
{
    x++; // Increments x by 1
    y++; // Increments x by 1
    std::cout << x << '\n';
    std::cout << y << '\n';
}

int main()
{
    int x{6};
    int y{7};

    printValues(x, y); // This function call has two arguments, 6 and 7
    std::cout << x << '\n';
    std::cout << y << '\n';

    return 0;
}
```

This results in the output is:

```
7
8
6
7
```

Which shows how the variables x and y are unchanged.

5.4.2 Using return values as arguments

It is possible, and good practice, to use return values as arguments:

```cpp
#include <iostream>

int getValueFromUser()
{
  std::cout << "Enter an integer: ";
  int input{};
  std::cin >> input;

  return input;
}

void printDouble(int value)
```

```cpp
{
  std::cout << value << " doubled is: " << value * 2 << '\n';
}

int main()
{
  printDoubleL(getValueFromUser()); // Using return value from
      getValueFromUser() as a parameter to call printDouble(int).

  return 0;
}
```

5.5 Local scope

Variables defined inside the body of a function are called **local variables**. Function parameters are also generally considered to be local variables, and we will include them as such:

```cpp
int add(int x, int y) // function parameters x and y are local variables
{
    int z{ x + y }; // z is a local variable

    return z;
}
```

5.5.1 Local variable lifetime

Local variables are destroyed in the opposite order of creation at the end of the set of curly braces in which it is defined (or for a function parameter, at the end of the function).

```cpp
int add(int x, int y)
{
    int z{ x + y };

    return z;
} // z, y, and x destroyed here
```

What happens when an object is destroyed? In most cases, nothing. The destroyed object simply becomes invalid. Any use of an object after it has been destroyed will result in undefined behavior. At some point after destruction, the memory used by the object will be deallocated (freed up for reuse).

An identifier's scope determines where the identifier can be seen and used within the source code. When an identifier can be seen and used, we say it is in scope. When an identifier can not be seen, we can not use it, and we say it is out of scope. Scope is a compile-time property, and trying to use an identifier when it is not in scope will result in a compile error.
The identifier of a local variable has local scope.

Scope: An identifier with local scope (technically called **block scope**) is usable from the point of definition to the end of the innermost pair of curly braces containing the identifier.

Here's a program demonstrating the scope of a variable named `x`:

```cpp
#include <iostream>

// x is not in scope anywhere in this function
void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    // x can not be used here because it's not in scope yet

    int x{ 0 }; // x enters scope here and can now be used within this
        function

    doSomething();

    return 0;
} // x goes out of scope here and can no longer be used
```

5.5.2 Where to define local variables

In modern C++, the best practice is that local variables inside the function body should be defined as close to their first use as reasonable:

```cpp
  std::cout << "Enter an integer: ";
  int x{};        // x defined here
  std::cin >> x; // and used here

  std::cout << "Enter another integer: ";
  int y{};        // y defined here
  std::cin >> y; // and used here
```

5.5.3 Introduction to temporary objects

A temporary object (also sometimes called an anonymous object) is an unnamed object that is used to hold a value that is only needed for a short period of time. Temporary objects are generated by the compiler when they are needed.
There are many different ways that temporary values can be created, but here's a common one:

```cpp
#include <iostream>

int getValueFromUser()
{
  std::cout << "Enter an integer: ";
  int input{};
  std::cin >> input;

  return input; // return the value of input back to the caller
}

int main()
{
  std::cout << getValueFromUser() << '\n'; // where does the returned
      value get stored?

  return 0;
}
```

In the above program, the function getValueFromUser() returns the value stored in local variable input back to the caller. Because input will be destroyed at the end of the function, the caller receives a copy of the value so that it has a value it can use even after input is destroyed.
But where is the value that is copied back to the caller stored? We haven't defined any variables in main(). The answer is that the return value is stored in a temporary object. This temporary object is then passed to std::cout to be printed.

5.6 Function overloading

A forward declaration allows us to tell the compiler about the existence of an identifier before actually defining the identifier.
In the case of functions, this allows us to tell the compiler about the existence of a function before we define the function's body. This way, when the compiler encounters a call to the function, it'll understand that we're making a function call, and can check to ensure we're calling the function correctly, even if it doesn't yet know how or where the function is defined.

Declaration

```cpp
int add(int x, int y); // function declaration includes return type, name, parameters, and semicolon.  No function body!
```

You may be wondering why we would use a forward declaration if we could just reorder the functions to make our programs work.

Most often, forward declarations are used to tell the compiler about the existence of some function that has been defined in a different code file. Reordering isn't possible in this scenario because the caller and the callee are in completely different files!

Forward declarations can also be used to define our functions in an order-agnostic manner. This allows us to define functions in whatever order maximizes organization (e.g. by clustering related functions together) or reader understanding.

Less often, there are times when we have two functions that call each other. Reordering isn't possible in this case either, as there is no way to reorder the functions such that each is before the other. Forward declarations give us a way to resolve such circular dependencies.

Forgetting the function body

New programmers often wonder what happens if they forward declare a function but do not define it.
The answer is: it depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made and the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can't resolve the function call.

5.6.1 Declarations vs. definitions

A **declaration** tells the compiler about the existence of an identifier and its associated type information. Here are some examples of declarations:

```cpp
int add(int x, int y); // tells the compiler about a function named "add" that takes two int parameters and returns an int.  No body!
int x;                 // tells the compiler about an integer variable named x
```

A **definition** is a declaration that actually implements (for functions and types) or instantiates (for variables) the identifier. Here are some examples of definitions:

```cpp
// because this function has a body, it is an implementation of function add()
int add(int x, int y)
{
    int z{ x + y };   // instantiates variable z

    return z;
}

int x;                // instantiates variable x
```

5.6.2 The one definition rule (ODR)

The one definition rule (or ODR for short) is a well-known rule in C++. The ODR has three parts:

1. Within a file, each function, variable, type, or template in a given scope can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.

2. Within a program, each function or variable in a given scope can only have one definition. This rule exists because programs can have more than one file (we'll cover this in the next lesson). Functions and variables not visible to the linker are excluded from this rule (discussed further in lesson 7.6 – Internal linkage).

3. Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical. We haven't covered what most of these things are yet, so don't worry about this for now – we'll bring it back up when it's relevant.

Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will cause the linker to issue a redefinition error. Violating ODR part 3 will cause undefined behavior.

Chapter 6

Constants

In programming, a **constant** is a value that may not be changed during the program's execution.
C++ supports two different kinds of constants:

- **Named constants** (or **constant variables** are constant values that are associated with an identifier. These are also sometimes called symbolic constants.
- **Literal constants** are constant values that are not associated with an identifier.

6.1 Constant Variables

Declaration

To declare a constant variable, we place the const keyword (called a "const qualifier") adjacent to the object's type:

```cpp
const double gravity { 9.8 };  // preferred use of const before type
int const sidesInSquare { 4 }; // "east const" style, okay but not
    preferred
```

Initialization

Const variables must be initialized when you define them since their value cannot be changed afterwards.

```cpp
const double gravity; // error: const variables must be initialized
gravity = 9.9;        // error: const variables can not be changed
```

Note that const variables can be initialized from other variables (including non-const ones):

```cpp
int age{};
std::cin >> age;

const int constAge { age }; // initialize const variable using non-const
    value

age = 5;       // ok: age is non-const, so we can change its value
constAge = 6; // error: constAge is const, so we cannot change its value
```

Naming convention

because const variables act like normal variables (except they can not be assigned to), there is no reason that they need a special naming convention. For this reason, we prefer using the same naming convention that we use for non-const variables (e.g. earthGravity).

Const function parameters

Function parameters can be made constants via the const keyword:

```
void printInt(const int x)
{
    std::cout << x << '\n';
}
```

Note that we did not provide an explicit initializer for our const parameter x – the value of the argument in the function call will be used as the initializer for x.

Making a function parameter constant enlists the compiler's help to ensure that the parameter's value is not changed inside the function.

Note: Like in this case, it is useless to make parameters passed by value constant because while passing by value we are just making a copy of that variable.

Note: A function's return value may also be made const: However this is useless since the const qualifier on a return type is simply ignored, because they are temporary copies that will be destroyed anyway.

Why variables should be made constant

If a variable can be made constant, it generally should be made constant. This is important for a number of reasons:

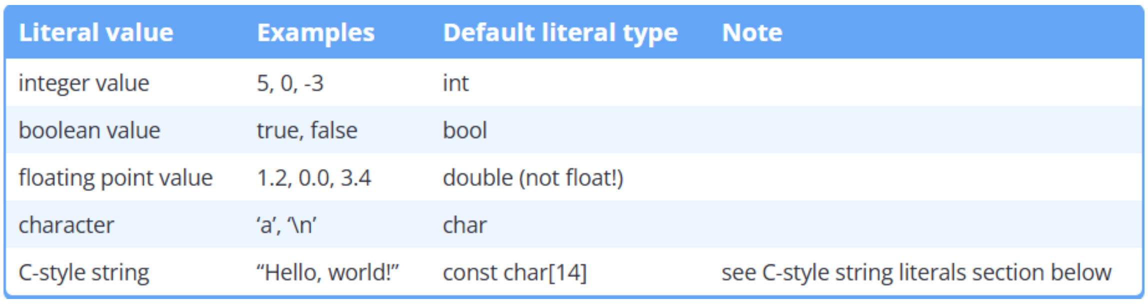
- It reduces the chance of bugs. By making a variable constant, you ensure that the value can't be changed accidentally.
- It provides more opportunity for the compiler to optimize programs. When the compiler can assume a value isn't changing, it is able to leverage more techniques to optimize the program, resulting in a compiled program that is smaller and faster. We'll discuss this further later in this chapter.
- Most importantly, it reduces the overall complexity of our programs. When trying to determine what a section of code is doing or trying to debug an issue, we know that a const variable can't have its value changed, so we don't have to worry about whether its value is actually changing, what value it is changing to, and whether that new value is correct.

6.2 Constant Literals

Literals are values that are inserted directly into the code. For example:

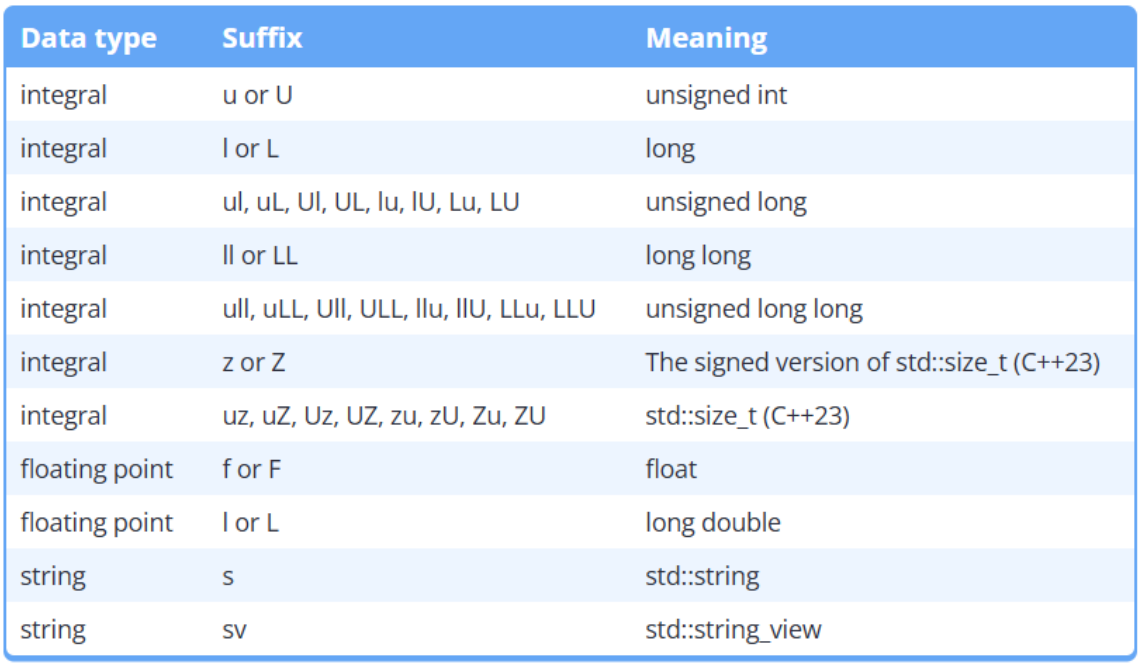
```
return 5;                      // 5 is an integer literal
bool myNameIsAlex { true };    // true is a boolean literal
double d { 3.4 };              // 3.4 is a double literal
std::cout << "Hello, world!"; // "Hello, world!" is a C-style string
    literal
```

Just like objects have a type, all literals have a type. The type of a literal is deduced from the literal's value. For example, a literal that is a whole number (e.g. 5) is deduced to be of type int.



Literal value	Examples	Default literal type	Note
integer value	5, 0, -3	int	
boolean value	true, false	bool	
floating point value	1.2, 0.0, 3.4	double (not float!)	
character	'a', '\n'	char	
C-style string	"Hello, world!"	const char[14]	see C-style string literals section below

If the default type of a literal is not as desired, you can change the type of a literal by adding a suffix. Here are some of the more common suffixes:



Data type	Suffix	Meaning
integral	u or U	unsigned int
integral	l or L	long
integral	ul, uL, Ul, UL, lu, lU, Lu, LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz, uZ, Uz, UZ, zu, zU, Zu, ZU	std::size_t (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	std::string
string	sv	std::string_view

6.3 The as-if rule and compile-time optimization

In programming, optimization is the process of modifying software to make it work more efficiently (e.g. to run faster, or use fewer resources). Optimization can have a huge impact on the overall performance level of an application.

Some types of optimization are typically done by hand. A program called a profiler can be used to see how long various parts of the program are taking to run, and which are impacting overall performance.

Other kinds of optimization can be performed automatically. A program that optimizes another program is called an optimizer. Optimizers typically work at a low-level, looking for ways to improve statements or expressions by rewriting, reordering, or eliminating them.

6.3.1 The as-if rule

In C++, compilers are given a lot of leeway to optimize programs.

As-if rule: The compiler can modify a program however it likes in order to produce more optimized code, so long as those modifications do not affect a program's "observable behavior".

Modern compilers employ a variety of different techniques in order to optimize a program effectively. Which techniques can be applied depends on the program and the quality of the compiler and optimizer.

6.3.2 An optimization opportunity

Consider the following short program:

```cpp
#include <iostream>

int main()
{
  int x { 3 + 4 };
  std::cout << x << '\n';

  return 0;
}
```

If this program were compiled exactly as it was written (with no optimizations), the compiler would generate an executable that calculates the result of 3 + 4 at runtime (when the program is run). If the program were

executed a million times, 3 + 4 would be evaluated a million times, and the resulting value of 7 produced a million times.
Because the result of 3 + 4 never changes (it is always 7), re-calculating this result every time the program is run is wasteful.

6.3.3 Compile-time evaluation

Modern C++ compilers are capable of fully or partially evaluating certain expressions at compile-time (rather than at runtime). When the compiler fully or partially evaluates an expression at compile-time, this is called compile-time evaluation.

One of the original forms of compile-time evaluation is called "**constant folding**". Constant folding is an optimization technique where the compiler replaces expressions that have literal operands with the result of the expression. Using constant folding, the compiler would recognize that the expression 3 + 4 has constant operands, and then replace the expression with the result 7.

The result would be equivalent to the following:

```cpp
#include <iostream>

int main()
{
  int x { 7 };
  std::cout << x << '\n';

  return 0;
}
```

There are other types of compile-time optimization such as: **constant propagation**, **dead code elimination**...

6.3.4 Compile-time constants and runtime constants

Constants in C++ are sometimes divided into two informal categories.
A **compile-time constant** is a constant whose value is known at compile-time. Examples include:

- Literals.
- Constant objects whose initializers are compile-time constants.

A **runtime constant** is a constant whose value is determined in a runtime context. Examples include:

- Constant function parameters.
- Constant objects whose initializers are non-constants or runtime constants.

6.4 Constant Expressions

By default, expressions evaluate at runtime. And in some cases, they must do so:

```cpp
std::cin >> x;
std::cout << 5 << '\n';
```

Because input and output can't be performed at compile time, the expressions above are required to evaluate at runtime.

However, the compiler can optimize programs by shifting work from runtime to compile-time. Under the as-if rule, the compiler may choose whether to evaluate certain expressions at runtime or compile-time:

```cpp
const double x { 1.2 };
const double y { 3.4 };
const double z { x + y }; // x + y may evaluate at runtime or compile-time
```

The expression x + y would normally evaluate at runtime, but since the value of x and y are known at compile-time, the compiler may opt to perform compile-time evaluation instead and initialize z with the compile-time calculated value 4.6.

6.4.1 Constexpr

We can enlist the compiler's help to ensure we get a compile-time constant variable where we desire one. To do so, we use the constexpr keyword (which is shorthand for "constant expression") instead of const in a variable's declaration. A constexpr variable is always a compile-time constant. As a result, a constexpr variable must be initialized with a constant expression, otherwise a compilation error will result.

- `const` means that the value of an object cannot be changed after initialization. The value of the initializer may be known at compile-time or runtime. The const object can be evaluated at runtime.
- `constexpr` means that the object can be used in a constant expression. The value of the initializer must be known at compile-time. The constexpr object can be evaluated at runtime or compile-time.

Constexpr variables are implicitly const. Const variables are not implicitly constexpr (except for const integral variables with a constant expression initializer). Although a variable can be defined as both constexpr and const, in most cases this is redundant, and we only need to use either const or constexpr.
Unlike const, constexpr is not part of an object's type. Therefore a variable defined as constexpr int actually has type const int (due to the implicit const that constexpr provides for objects).

6.4.2 constexpr functions

6.4.3 The benefits of compile-time programming

That same code compiled on a different platform, or with a different compiler, or using different compilation options, or slightly modified, may produce a different result. Because the as-if rule is applied invisibly to us, we get no feedback from the compiler on what portions of code it decided to evaluate at compile-time, or why. Code we desire to be evaluated at compile-time may not even be eligible (due to a typo or misunderstanding), and we may never know.

To improve upon this situation, the C++ language provides ways for us to be explicit about what parts of code we want to execute at compile-time. The use of language features that result in compile-time evaluation is called compile-time programming.

These features can help improve software in a number of areas:

- **Performance**: Compile-time evaluation makes our programs smaller and faster. The more code we can ensure is capable of evaluating at compile-time, the more performance benefit we'll see.
- **Versatility**: We can always use such code in places that require a compile-time value. Code that relies on the as-if rule to evaluate at compile-time can't be used in such places (even if the compiler opts to evaluate that code at compile-time) – this decision was made so that code that compiles today won't stop compiling tomorrow, when the compiler decides to optimize differently.
- **Predictability**: We can have the compiler halt compilation if it determines that code cannot be executed at compile-time (rather than silently opting to have that code evaluate at runtime instead). This allows us to ensure a section of code we really want to execute at compile-time will.
- **Error Detection**: We can have the compiler reliably detect certain kinds of programming errors at compile-time, and halt the build if it encounters them. This is much more effective than trying to detect and gracefully handle those same errors at runtime.
- **Safety**: Perhaps most importantly, undefined behavior is not allowed at compile-time. If we do something that causes undefined behavior at compile-time, the compiler should halt the build and ask us to fix it. Note that this is a hard problem for compilers, and they may not catch all cases.

6.5 Introduction to std::string

While C-style string literals are fine to use, C-style string variables behave oddly, are hard to work with and are dangerous. In modern C++, C-style string variables are best avoided.

Fortunately, C++ has introduced two additional string types into the language that are much easier and safer to work with: std::string and `std::string_view` (C++17). Unlike the types we've introduced previously, std::string and `std::string_view` aren't fundamental types (they're class types, which we'll cover in the future). However, basic usage of each is straightforward and useful enough that we'll introduce them here.

Declaration

```cpp
#include <string> // allows use of std::string

int main()
{
    std::string name {}; // empty string

    return 0;
}
```

Initialization and Assignment

```cpp
    std::string name { "Alex" }; // initialize name with string literal
        "Alex"
    name = "John";               // change name to "John"
```

String output with std::cout

std::string objects can be output as expected using std::cout:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name { "Alex" };
    std::cout << "My name is: " << name << '\n';

    return 0;
}
```

String input with std::cin

Using std::string with std::cin may yield some surprises!

```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::cin >> name; // this won't work as expected since std::cin breaks
        on whitespace

    std::cout << "Enter your favorite color: ";
    std::string color{};
    std::cin >> color;

    std::cout << "Your name is " << name << " and your favorite color is "
        << color << '\n';

    return 0;
}
```

Here's the results from a sample run of this program:

```
Enter your full name: John Doe
Enter your favorite color: Your name is John and your favorite color is Doe
```

There are some workarounds such as `std::getline()` or `std::ws` but we will not be diving into that.

The length of a std::string

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name{ "Alex" };
    std::cout << name << " has " << name.length() << " characters\n";

    return 0;
}
```

This prints:

```
Alex has 4 characters
```

Note that instead of asking for the string length as length(name), we say name.length(). The length() function isn't a normal standalone function – it's a special type of function that is nested within std::string called a member function. Because the length() member function is declared inside of std::string, it is sometimes written as std::string::length() in documentation.
We'll cover member functions, including how to write your own, in more detail later.

Initializing a std::string is expensive

Whenever a std::string is initialized, a copy of the string used to initialize it is made. Making copies of strings is expensive, so care should be taken to minimize the number of copies made.

Returning a std::string

When a function returns by value to the caller, the return value is normally copied from the function back to the caller. So you might expect that you should not return std::string by value, as doing so would return an expensive copy of a std::string.
However, as a rule of thumb, it is okay to return a std::string by value when the expression of the return statement resolves to any of the following:

- A local variable of type std::string.
- A std::string that has been returned by value from another function call or operator.
- A std::string temporary that is created as part of the return statement.

Note: If you try to define a constexpr std::string, your compiler will probably generate an error: This happens because constexpr std::string isn't supported at all in C++17 or earlier, and only works in very limited cases in C++20/23. If you need constexpr strings, use `std::string_view` instead

6.6 Introduction to `std::string_view`

Chapter 7

Scope, duration, and linkage

7.1 Compound statements (blocks)

A compound statement (also called a block, or block statement) is a group of zero or more statements that is treated by the compiler as if it were a single statement.
Blocks begin with a symbol, end with a symbol, with the statements to be executed being placed in between. Blocks can be used anywhere a single statement is allowed. No semicolon is needed at the end of a block.
You have already seen an example of blocks when writing functions, as the function body is a block:

```cpp
int add(int x, int y)
{ // start block
    return x + y;
} // end block (no semicolon)

int main()
{ // start block

    // multiple statements
    int value {}; // this is initialization, not a block
    add(3, 4);

    return 0;

} // end block (no semicolon)
```

Blocks can be nested inside other blocks:

```cpp
int add(int x, int y)
{ // block
    return x + y;
} // end block

int main()
{ // outer block

    // multiple statements
    int value {};

    { // inner/nested block
        add(3, 4);
    } // end inner/nested block

    return 0;

} // end outer block
```

One of the most common use cases for blocks is in conjunction with `if statements`. By default, an `if statements` executes a single statement if the condition evaluates to `true`. However, we can replace this single statement with a block of statements if we want multiple statements to execute when the condition evaluates to `true`. We will come back to `if statements` later.

7.2 Local variables

We already introduced local variables, which are variables that are defined inside a function (including function parameters).

It turns out that C++ actually doesn't have a single attribute that defines a variable as being a local variable. Instead, local variables have several different properties that differentiate how these variables behave from other kinds of (non-local) variables. We'll explore these properties in this and upcoming lessons.

We also introduced the concept of scope. An identifier's scope determines where an identifier can be accessed within the source code. When an identifier can be accessed, we say it is in scope. When an identifier can not be accessed, we say it is out of scope. Scope is a compile-time property, and trying to use an identifier when it is out of scope will result in a compile error.

7.2.1 Local variables have block scope

Local variables have block scope, which means they are in scope from their point of definition to the end of the block they are defined within.

```
int main()
{
    int i { 5 }; // i enters scope here
    double d { 4.0 }; // d enters scope here

    return 0;
} // d and i go out of scope here
```

All variable names within a scope must be unique

Local variables in nested blocks

```
int main() // outer block
{
    int x { 5 }; // x enters scope and is created here

    { // nested block
        int y { 7 }; // y enters scope and is created here
    } // y goes out of scope and is destroyed here

    // y can not be used here because it is out of scope in this block

    return 0;
} // x goes out of scope and is destroyed here
```

Note that nested blocks are considered part of the scope of the outer block in which they are defined. Consequently, variables defined in the outer block can be seen inside a nested block.

Local variables have no linkage

Identifiers have another property named linkage. An identifier's linkage determines whether a declaration of that same identifier in a different scope refers to the same object (or function).
Local variables have no linkage. Each declaration of an identifier with no linkage refers to a unique object or function.
For example:

```
int main()
{
    int x { 2 }; // local variable, no linkage

    {
        int x { 3 }; // this declaration of x refers to a different object
            than the previous x
        // Note that as long as we are in this block only x = 3 is
            accessible
```

```cpp
    }

    return 0;
}
```

Scope and linkage may seem somewhat similar. However, scope determines where declaration of a single identifier can be seen and used in the code. Linkage determines whether multiple declarations of the same identifier refer to the same object or not.

If a variable is only used within a nested block, it should be defined inside that nested block:

Variables should be defined in the most limited scope

```cpp
#include <iostream>

int main()
{
    // do not define y here

    {
        // y is only used inside this block, so define it here
        int y { 5 };
        std::cout << y << '\n';
    }

    // otherwise y could still be used here, where it's not needed

    return 0;
}
```

By limiting the scope of a variable, you reduce the complexity of the program because the number of active variables is reduced. Further, it makes it easier to see where variables are used (or aren't used). A variable defined inside a block can only be used within that block (or nested blocks). This can make the program easier to understand.

7.3 Global variables

In C++, variables can also be declared outside of a function. Such variables are called **global variables**. By convention, global variables are declared at the top of a file, below the includes, in the global namespace. Here's an example of a global variable being defined:

```cpp
#include <iostream>

// Variables declared outside of a function are global variables
int g_x {}; // global variable g_x

void doSomething()
{
    // global variables can be seen and used everywhere in the file
    g_x = 3;
    std::cout << g_x << '\n';
}

int main()
{
    doSomething();
    std::cout << g_x << '\n';

    // global variables can be seen and used everywhere in the file
    g_x = 5;
    std::cout << g_x << '\n';

    return 0;
}
// g_x goes out of scope here
```

Note: Unlike local variables, which are uninitialized by default, variables with static duration are zero-initialized by default.

Constant global variables

Just like local variables, global variables can be constant. As with all constants, constant global variables must be initialized.

```cpp
const int g_x;     // error: constant variables must be initialized
constexpr int g_w; // error: constexpr variables must be initialized

const int g_y { 1 };      // const global variable g_y, initialized with a
    value
constexpr int g_z { 2 }; // constexpr global variable g_z, initialized
    with a value
```

Avoid Global Variables

New programmers are often tempted to use lots of global variables, because they can be used without having to explicitly pass them to every function that needs them. However, use of non-constant global variables should generally be avoided altogether!

7.4 Variable shadowing

Each block defines its own scope region. So what happens when we have a variable inside a nested block that has the same name as a variable in an outer block? When this happens, the nested variable “hides” the outer variable in areas where they are both in scope. This is called **name hiding** or **shadowing**.

7.4.1 Shadowing of local variables

```cpp
#include <iostream>

int main()
{ // outer block
    int apples { 5 }; // here's the outer block apples

    { // nested block
        // apples refers to outer block apples here
        std::cout << apples << '\n'; // print value of outer block apples

        int apples{ 0 }; // define apples in the scope of the nested block

        // apples now refers to the nested block apples
        // the outer block apples is temporarily hidden

        apples = 10; // this assigns value 10 to nested block apples, not
            outer block apples

        std::cout << apples << '\n'; // print value of nested block apples
    } // nested block apples destroyed


    std::cout << apples << '\n'; // prints value of outer block apples

    return 0;
} // outer block apples destroyed
```

In the above program, we first declare a variable named apples in the outer block. This variable is visible within the inner block, which we can see by printing its value (5). Then we declare a different variable (also named apples) in the nested block. From this point to the end of the block, the name apples refers to the nested block apples, not the outer block apples.

Thus, when we assign value 10 to apples, we're assigning it to the nested block apples. After printing this value (10), the nested block ends and nested block apples is destroyed. The existence and value of outer block apples is not affected, and we prove this by printing the value of outer block apples (5).
Note that if the nested block apples had not been defined, the name apples in the nested block would still refer to the outer block apples, so the assignment of value 10 to apples would have applied to the outer block apples:

7.4.2 Shadowing of global variables

Similar to how variables in a nested block can shadow variables in an outer block, local variables with the same name as a global variable will shadow the global variable wherever the local variable is in scope.

Avoid variable shadowing

Shadowing of local variables should generally be avoided: It can lead to inadvertent errors where the wrong variable is used or modified. Some compilers will issue a warning when a variable is shadowed.

7.5 Internal linkage

An identifier with **internal linkage** can be seen and used within a single translation unit, but it is not accessible from other translation units.

An identifier's linkage determines whether other declarations of that name refer to the same object or not. Local variables in C++ have no linkage.

Global variables and function identifiers can have either **internal linkage** or **external linkage**.

7.5.1 Global Variables Linkage

Non-constant globals have external linkage by default, but can be given internal linkage via the static keyword.

```
#include <iostream>

static int g_x{}; // non-constant globals have external linkage by
    default, but can be given internal linkage via the static keyword

const int g_y{ 1 }; // const globals have internal linkage by default
constexpr int g_z{ 2 }; // constexpr globals have internal linkage by
    default

int main()
{
    std::cout << g_x << ' ' << g_y << ' ' << g_z << '\n';
    return 0;
}
```

Global variables with internal linkage are sometimes called **internal variables**.

Note: Const and constexpr global variables have internal linkage by default (and thus don't need the static keyword – if it is used, it will be ignored).

7.5.2 Functions with internal linkage

Function identifiers also have linkage. Functions default to external linkage (which we'll cover in the next lesson), but can be set to internal linkage via the static keyword:

```
static int add(int x, int y)
{
    return x + y;
}
```

The one-definition rule and internal linkage

We noted that the one-definition rule says that an object or function can't have more than one definition, either within a file or a program.

However, it's worth noting that internal objects (and functions) that are defined in different files are considered to be independent entities (even if their names and types are identical), so there is no violation of the one-definition rule. Each internal object only has one definition.

7.6 External linkage

An identifier with **external linkage** can be seen and used both from the file in which it is defined, and from other code files (via a forward declaration). In this sense, identifiers with external linkage are truly "global" in that they can be used anywhere in your program!

Functions have external linkage by default

Global variables with external linkage

Global variables with external linkage are sometimes called external variables. To make a global variable external (and thus accessible by other files), we can use the extern keyword to do so:

```cpp
int g_x { 2 }; // non-constant globals are external by default (no need to use extern)

extern const int g_y { 3 }; // const globals can be defined as extern, making them external

int main()
{
    return 0;
}
```

7.6.1 Variable forward declarations

To actually use an external global variable that has been defined in another file, you also must place a forward declaration for the global variable in any other files wishing to use the variable. For variables, creating a forward declaration is also done via the extern keyword (with no initialization value).

Here is an example of using variable forward declarations:
a.cpp

```cpp
// global variable definitions
int g_x { 2 };              // non-constant globals have external linkage by default
extern const int g_y { 3 }; // this extern gives g_y external linkage
```

main.cpp

```cpp
#include <iostream>

extern int g_x;       // this extern is a forward declaration of a variable named g_x that is defined somewhere else
extern const int g_y; // this extern is a forward declaration of a const variable named g_y that is defined somewhere else

int main()
{
    std::cout << g_x << ' ' << g_y << '\n'; // prints 2 3

    return 0;
}
```

Note that the extern keyword has different meanings in different contexts. In some contexts, extern means "give this variable external linkage". In other contexts, extern means "this is a forward declaration for an external variable that is defined somewhere else".

Note: Function forward declarations don't need the extern keyword.

7.7 Inline functions and variables

Consider the case where you need to write some code to perform some discrete task, like reading input from the user, or outputting something to a file, or calculating a particular value. When implementing this code, you essentially have two options:

- Write the code as part of an existing function (called writing code "in-place" or "inline").
- Create a new function (and possibly sub-functions) to handle the task.

Putting the code in a new function provides many potential benefits, as small functions:

- Are easier to read and understand in the context of the overall program.
- Are easier to reuse, as functions are naturally modular.
- Are easier to update, as the code only needs to be modified in one place.

However, one downside of using a new function is that every time a function is called, there is a certain amount of performance overhead that occurs. Consider the following example:

```
#include <iostream>

int min(int x, int y)
{
    return (x < y) ? x : y;
}

int main()
{
    std::cout << min(5, 6) << '\n';
    std::cout << min(3, 2) << '\n';
    return 0;
}
```

When a call to min() is encountered, the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with the values of various CPU registers (so they can be restored upon returning). Then parameters x and y must be instantiated and then initialized. Then the execution path has to jump to the code in the min() function. When the function ends, the program has to jump back to the location of the function call, and the return value has to be copied so it can be output. This has to be done for each function call. All of the extra work that must happen to setup, facilitate, and/or cleanup after some task (in this case, making a function call) is called **overhead**.

7.7.1 Inline expansion

Fortunately, the C++ compiler has a trick that it can use to avoid such overhead cost: Inline expansion is a process where a function call is replaced by the code from the called function's definition.

For example, if the compiler expanded the min() calls in the above example, the resulting code would look like this:

```
#include <iostream>

int main()
{
    std::cout << ((5 < 6) ? 5 : 6) << '\n';
    std::cout << ((3 < 2) ? 3 : 2) << '\n';
    return 0;
}
```

However, inline expansion has its own potential cost: if the body of the function being expanded takes more instructions than the function call being replaced, then each inline expansion will cause the executable to grow larger. Larger executables tend to be slower (due to not fitting as well in memory caches).

The decision about whether a function would benefit from being made inline (because removal of the function call overhead outweighs the cost of a larger executable) is not straightforward. Inline expansion could result in performance improvements, performance reductions, or no change to performance at all, depending on the relative cost of a function call, the size of the function, and what other optimizations can be performed.

Inline expansion is best suited to simple, short functions (e.g. no more than a few statements), especially cases where a single function call can be executed more than once (e.g. function calls inside a loop).

7.7.2 Inline keyword in old C++

However, in modern C++, the inline keyword is no longer used to request that a function be expanded inline. There are quite a few reasons for this:

- Using inline to request inline expansion is a form of premature optimization, and misuse could actually harm performance.
- The inline keyword is just a hint to help the compiler determine where to perform inline expansion. The compiler is completely free to ignore the request, and it may very well do so. The compiler is also free to perform inline expansion of functions that do not use the inline keyword as part of its normal set of optimizations.
- The inline keyword is defined at the wrong level of granularity. We use the inline keyword on a function definition, but inline expansion is actually determined per function call. It may be beneficial to expand some function calls and detrimental to expand others, and there is no syntax to influence this.

Modern optimizing compilers are typically good at determining which function calls should be made inline – better than humans in most cases. As a result, the compiler will likely ignore or devalue any use of inline to request inline expansion for your functions.

7.7.3 Inline keyword in modern C++

In modern C++, the term inline has evolved to mean “multiple definitions are allowed”. Thus, an inline function is one that is allowed to be defined in multiple translation units (without violating the ODR).

Inline functions have two primary requirements:

The compiler needs to be able to see the full definition of an inline function in each translation unit where the function is used (a forward declaration will not suffice on its own). Only one such definition can occur per translation unit, otherwise a compilation error will occur. The definition can occur after the point of use if a forward declaration is also provided. However, the compiler will likely not be able to perform inline expansion until it has seen the definition (so any uses between the declaration and definition will probably not be candidates for inline expansion). Every definition for an inline function (with external linkage, which functions have by default) must be identical, otherwise undefined behavior will result.

Why not make all functions inline and defined in a header file?

Mainly because doing so can increase your compile times significantly.

When a header containing an inline function is `#included` into a source file, that function definition will be compiled as part of that translation unit. An inline function `#included` into 6 translation units will have its definition compiled 6 times (before the linker deduplicates the definitions). Conversely, a function defined in a source file will have its definition compiled only once, no matter how many translation units its forward declaration is included into.

Second, if a function defined in a source file changes, only that single source file needs to be recompiled. When an inline function in a header file changes, every code file that includes that header (either directly or via another header) needs to recompiled. On large projects, this can cause a cascade of recompilation and have a drastic impact.

7.7.4 Inline variables

C++17 introduces **inline variables**, which are variables that are allowed to be defined in multiple files. Inline variables work similarly to inline functions, and have the same requirements (the compiler must be able to see an identical full definition everywhere the variable is used).

```
inline int a{0};
```

7.8 Sharing global constants across multiple files

In some applications, certain symbolic constants may need to be used throughout your code (not just in one location). These can include physics or mathematical constants that don't change (e.g. pi or Avogadro's number), or application-specific "tuning" values (e.g. friction or gravity coefficients). Instead of redefining these constants in every file that needs them (a violation of the "Don't Repeat Yourself" rule), it's better to declare them once in a central location and use them wherever needed. That way, if you ever need to change them, you only need to change them in one place, and those changes can be propagated out.
This lesson covers the most common ways to do this.

7.8.1 Constants Header File

Prior to C++17, the following is the easiest and most common solution:

- Create a header file to hold these constants
- Inside this header file, define a namespace (discussed in lesson 7.2 – User-defined namespaces and the scope resolution operator)
- Add all your constants inside the namespace (make sure they're constexpr)
- `#include` the header file wherever you need it

`constants.h`

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// Define your own namespace to hold constants
namespace constants
{
    // Global constants have internal linkage by default
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on
        this planet
    // ... other related constants
}
#endif
```

main.cpp:

```
#include "constants.h" // include a copy of each constant in this file

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi <<
        '\n';

    return 0;
}
```

While this is simple (and fine for smaller programs), every time constants.h gets `#included` into a different code file, each of these variables is copied into the including code file. Therefore, if constants.h gets included into 20 different code files, each of these variables is duplicated 20 times. Header guards won't stop this from happening, as they only prevent a header from being included more than once into a single including file, not from being included one time into multiple different code files. This introduces two challenges:

- Changing a single constant value would require recompiling every file that includes the constants header, which can lead to lengthy rebuild times for larger projects.
- If the constants are large in size and can't be optimized away, this can use a lot of memory.

7.8.2 External variables

If you're actively changing values or adding new constants, the prior solution might be problematic, at least until things settle.
One way to avoid these problems is by turning these constants into external variables, since we can then have a single variable (initialized once) that is shared across all files. In this method, we'll define the constants in a .cpp file (to ensure the definitions only exist in one place), and put forward declarations in the header (which will be included by other files).
constants.cpp

```cpp
#include "constants.h"

namespace constants
{
    // We use extern to ensure these have external linkage
    extern constexpr double pi { 3.14159 };
    extern constexpr double avogadro { 6.0221413e23 };
    extern constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is
        light on this planet
}
```

constants.h

```cpp
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace constants
{
    // Since the actual variables are inside a namespace, the forward
        declarations need to be inside a namespace as well
    // We can't forward declare variables as constexpr, but we can forward
        declare them as (runtime) const
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}

#endif
```

Now the symbolic constants will get instantiated only once (in constants.cpp) instead of in each code file where constants.h is `#included`, and all uses of these constants will be linked to the version instantiated in constants.cpp. Any changes made to constants.cpp will require recompiling only constants.cpp.

However, there are a couple of downsides to this method. First, because only the variable definitions are constexpr (the forward declarations aren't, and can't be), these constants are constant expressions only within the file they are actually defined in (constants.cpp). In other files, the compiler will only see the forward declaration, which doesn't define a constexpr value (and must be resolved by the linker). This means outside of the file where they are defined, these variables can't be used in a constant expression. Second, because constant expressions can typically be optimized more than runtime expressions, the compiler may not be able to optimize these as much.

7.8.3 Global constants as inline variables

In section 7.7 just covered inline variables, which are variables that can have more than one definition, so long as those definitions are identical. By making our constexpr variables inline, we can define them in a header

file and then include them into any .cpp file that requires them. This avoids both ODR violations and the downside of duplicated variables.
constants.h

```cpp
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    inline constexpr double pi { 3.14159 }; // note: now inline constexpr
    inline constexpr double avogadro { 6.0221413e23 };
    inline constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is
        light on this planet
    // ... other related constants
}
#endif
```

We can include constants.h into as many code files as we want, but these variables will only be instantiated once and shared across all code files.
This method does retain the downside of requiring every file that includes the constants header be recompiled if any constant value is changed.
Advantages:

- Can be used in constant expressions in any translation unit that includes them.
- Only one copy of each variable is required.

Downsides:

- Only works in C++17 onward.
- Changing anything in the header file requires recompiling files including the header.

7.9 Static local variables

The term static is one of the most confusing terms in the C++ language, in large part because static has different meanings in different contexts.

In prior lessons, we covered that global variables have static duration, which means they are created when the program starts and destroyed when the program ends.

We also discussed how the static keyword gives a global identifier internal-linkage, which means the identifier can only be used in the file in which it is defined.

Using the static keyword on a local variable changes its duration from automatic duration to static duration. This means the variable is now created at the start of the program, and destroyed at the end of the program (just like a global variable). As a result, the static variable will retain its value even after it goes out of scope!

```cpp
#include <iostream>

void incrementAndPrint()
{
    static int s_value{ 1 }; // static duration via static keyword.  This
        initializer is only executed once.
    ++s_value;
    std::cout << s_value << '\n';
} // s_value is not destroyed here, but becomes inaccessible because it
    goes out of scope

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

Best practice:
Const static local variables are generally okay to use.

Non-const static local variables should generally be avoided.

Chapter 8

Control flow

8.1 Introduction

When a program is run, the CPU begins execution at the top of main(), executes some number of statements (in sequential order by default), and then the program terminates at the end of main(). The specific sequence of statements that the CPU executes is called the program's execution path (or path, for short).

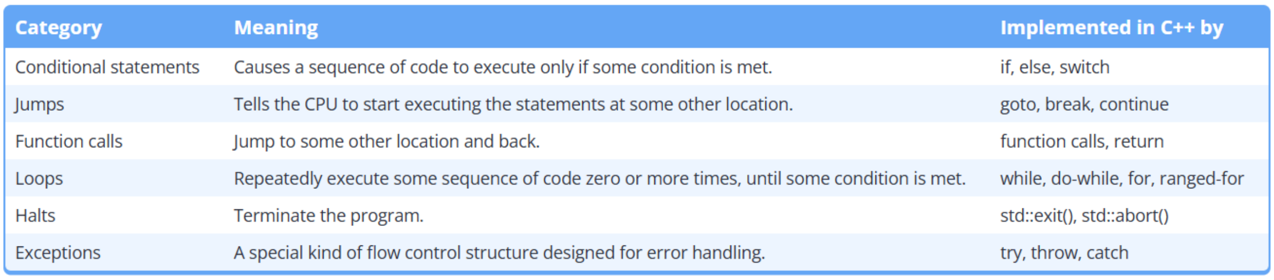
Consider the following program:

```cpp
1  #include <iostream>
2  
3  int main()
4  {
5      std::cout << "Enter an integer: ";
6  
7      int x{};
8      std::cin >> x;
9  
10     std::cout << "You entered " << x << '\n';
11 
12     return 0;
13 }
```

The execution path of this program includes lines 5, 7, 8, 10, and 12, in that order. This is an example of a straight-line program. Straight-line programs take the same path (execute the same statements in the same order) every time they are run.

However, often this is not what we desire. For example, if we ask the user for input, and the user enters something invalid, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program. In fact, the user may repeatedly enter invalid input, so the number of times we might need to ask them to make another selection isn't knowable until runtime.

C++ provides a number of different control flow statements (also called flow control statements), which are statements that allow the programmer to change the normal path of execution through the program.
The following table summarises categories of flow control statements:



Category	Meaning	Implemented in C++ by
Conditional statements	Causes a sequence of code to execute only if some condition is met.	if, else, switch
Jumps	Tells the CPU to start executing the statements at some other location.	goto, break, continue
Function calls	Jump to some other location and back.	function calls, return
Loops	Repeatedly execute some sequence of code zero or more times, until some condition is met.	while, do-while, for, ranged-for
Halts	Terminate the program.	std::exit(), std::abort()
Exceptions	A special kind of flow control structure designed for error handling.	try, throw, catch

The first category of control flow statements we'll talk about is **conditional statements**. A conditional statement is a statement that specifies whether some associated statement(s) should be executed or not.
C++ supports two basic kinds of conditionals: if statements and switch statements.

8.2 If statements

The most basic kind of conditional statement in C++ is the if statement. An if statement takes the form:

```cpp
if (condition)
    true_statement;
```

or with an optional else statement:

```cpp
if (condition)
    true_statement;
else
    false_statement;
```

Implicit blocks

If the programmer does not declare a block in the statement portion of an if statement or else statement, the compiler will implicitly declare one. Thus:

```cpp
#include <iostream>

int main()
{
    if (true)
        int x{ 5 };
    else
        int x{ 6 };

    std::cout << x << '\n';

    return 0;
}
```

This won't compile, with the compiler generating an error that identifier x isn't defined. This is because the above example is the equivalent of:

```cpp
#include <iostream>

int main()
{
    if (true)
    {
        int x{ 5 };
    } // x destroyed here
    else
    {
        int x{ 6 };
    } // x destroyed here

    std::cout << x << '\n'; // x isn't in scope here

    return 0;
}
```

Best practice:
Consider putting single statements associated with an if or else in blocks (particularly while you are learning). More experienced C++ developers sometimes disregard this practice in favor of tighter vertical spacing.

Implicit block should always be made explicit

```cpp
if (condition)
{
    true_statement;
}
else
```

```
{
    false_statement;
}
```

This avoid mistakes by making clear which statements are associated with the if statement and what are not.

8.2.1 If-else vs if-if

New programmers sometimes wonder when they should use if-else (if followed by one or more else-statements) or if-if (if followed by one or more additional if-statements).

- Use if-else when you only want to execute the code after the first true condition.
- Use if-if when you want to execute the code after all true conditions.

Here's a program that demonstrates this:

```cpp
#include <iostream>

void ifelse(bool a, bool b, bool c)
{
    if (a)      // always evaluates
        std::cout << "a";
    else if (b) // only evaluates when prior if-statement condition is
       false
        std::cout << "b";
    else if (c) // only evaluates when prior if-statement condition is
       false
        std::cout << "c";
    std::cout << '\n';
}

void ifif(bool a, bool b, bool c)
{
    if (a) // always evaluates
        std::cout << "a";
    if (b) // always evaluates
        std::cout << "b";
    if (c) // always evaluates
        std::cout << "c";
    std::cout << '\n';
}

int main()
{
    ifelse(false, true, true);
    ifif(false, true, true);

    return 0;
}
```

In the call to ifelse(false, true, true), a is false, so we do not execute the associated statement, and the associated else is executed instead. b is true, so we print b. Since this if condition was true, the associated else will not execute (and neither would any other else statements immediately following that one). Note that we only executed the code immediately after the first true condition (b).
In the call to ifif(false, true, true), a is false, so we do not execute the associated statement, and move to the next if. b is true, so we print b and move to the next if. c is true, so we print c. Note that we executed the code after all true conditions (b and c).
Now look at this somewhat similar function:

```cpp
char getFirstMatchingChar(bool a, bool b, bool c)
{
    if (a) // always evaluates
        return 'a';
    else if (b) // only evaluates when prior if-statement condition is
       false
        return 'b';
    else if (c) // only evaluates when prior if-statement condition is
        false
        return 'c';

    return 0;
}
```

Since we're using if-else, it's clear we only want to execute the code after the first true condition. However, in this case since we are inside a function, and every associated statement returns a value, we can just write this instead:

```
char getFirstMatchingChar(bool a, bool b, bool c)
{
    if (a) // always evaluates
        return 'a'; // returns when if-statement is true
    if (b) // only evaluates when prior if-statement condition is false
        return 'b'; // returns when if-statement is true
    if (c) // only evaluates when prior if-statement condition is false
        return 'c'; // returns when if-statement is true

    return 0;
}
```

8.3 Switch statements

8.4 While statements

8.5 Do while statements

8.6 For statements

8.7 Break and continue

8.8 Halts

The last category of flow control statement we'll cover in this chapter is the halt. A **halt** is a flow control statement that terminates the program. In C++, halts are implemented as functions (rather than keywords), so our halt statements will be function calls.
Let's take a brief detour, and recap what happens when a program exits normally. When the main() function returns (either by reaching the end of the function, or via a return statement), a number of different things happen.
First, because we're leaving the function, all local variables and function parameters are destroyed (as per usual).
Next, a special function called std::exit() is called, with the return value from main() (the status code) passed in as an argument.

8.8.1 The std::exit() function

std::exit() is a function that causes the program to terminate normally. Normal termination means the program has exited in an expected way. Note that the term normal termination does not imply anything about whether the program was successful. For example, let's say you were writing a program where you expected the user to type in a filename to process. If the user typed in an invalid filename, your program would probably return a non-zero status code to indicate the failure state, but it would still have a normal termination.
std::exit() performs a number of cleanup functions. First, objects with static storage duration are destroyed. Then some other miscellaneous file cleanup is done if any files were used. Finally, control is returned back to the OS, with the argument passed to std::exit() used as the status code.

Calling std::exit() explicitly

Although std::exit() is called implicitly after function main() returns, std::exit() can also be called explicitly to halt the program before it would normally terminate. When std::exit() is called this way, you will need to include the cstdlib header.

```cpp
#include <cstdlib> // for std::exit()
#include <iostream>

void cleanup()
{
    // code here to do any kind of cleanup required
    std::cout << "cleanup!\n";
}

int main()
{
    std::cout << 1 << '\n';
    cleanup();

    std::exit(0); // terminate and return status code 0 to operating system

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

std::exit() does not clean up local variables

Since does not clean up any local variables (either in the current function, or in functions up the call stack), calling std::exit() can be dangerous if your program relies on any local variables cleaning themselves up.

8.8.2 std::atexit()

Because std::exit() terminates the program immediately, you may want to manually do some cleanup before terminating. In this context, cleanup means things like closing database or network connections, deallocating any memory you have allocated, writing information to a log file, etc...
To assist with this, C++ offers the std::atexit() function, which allows you to specify a function that will automatically be called on program termination via std::exit().

8.8.3 std::abort and std::terminate

C++ contains two other halt-related functions.
The std::abort() function causes your program to terminate abnormally. Abnormal termination means the program had some kind of unusual runtime error and the program couldn't continue to run. For example, trying to divide by 0 will result in an abnormal termination. std::abort() does not do any cleanup.
The std::terminate() function is typically used in conjunction with exceptions (we'll cover exceptions in a later chapter). Although std::terminate can be called explicitly, it is more often called implicitly when an exception isn't handled (and in a few other exception-related cases). By default, std::terminate() calls std::abort().

When to use a halt

The short answer is "almost never". Destroying local objects is an important part of C++ (particularly when we get into classes), and none of the above-mentioned functions clean up local variables. Exceptions are a better and safer mechanism for handling error cases (And will be discussed in the next section).w

8.9 random Number Generation

The ability to generate random numbers can be useful in certain kinds of programs, particularly in games, statistical modelling programs, and cryptographic applications that need to encrypt and decrypt things. Take

games for example – without random events, monsters would always attack you the same way, you'd always find the same treasure, the dungeon layout would never change, etc... and that would not make for a very good game.
In real life, we often produce randomization by doing things like flipping a coin, rolling a dice, or shuffling a deck of cards. These events aren't actually random, but involve so many physical variables (e.g. gravity, friction, air resistance, momentum, etc...) that they become almost impossible to predict or control, and (unless you're a magician) produce results that are for all intents and purposes random.
However, computers aren't designed to take advantage of physical variables. Consequently, computers are generally incapable of generating truly random numbers (at least through software). Instead, modern programs typically simulate randomness using an algorithm.

8.9.1 Pseudo-random number generators (PRNGs)

To simulate randomness, programs typically use a pseudo-random number generator. A pseudo-random number generator (PRNG) is an algorithm that generates a sequence of numbers whose properties simulate a sequence of random numbers.
It's easy to write a basic PRNG algorithm. Here's a short PRNG example that generates 100 16-bit pseudo-random numbers:

```cpp
#include <iostream>

// For illustrative purposes only, don't use this
unsigned int LCG16() // our PRNG
{
    static unsigned int s_state{ 0 }; // only initialized the first time this function is called

    // Generate the next number

    // We modify the state using large constants and intentional overflow to make it hard
    // for someone to casually determine what the next number in the sequence will be.

    s_state = 8253729 * s_state + 2396403; // first we modify the state
    return s_state % 32768; // then we use the new state to generate the next number in the sequence
}

int main()
{
    // Print 100 random numbers
    for (int count{ 1 }; count <= 100; ++count)
    {
        std::cout << LCG16() << '\t';

        // If we've printed 10 numbers, start a new row
        if (count % 10 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

Each number appears to be pretty random with respect to the previous one.
As it turns out, this particular algorithm isn't very good as a random number generator (note how each result alternates between even and odd – that's not very random!). But most PRNGs work similarly to LCG16() – they just typically use more state variables and more complex mathematical operations in order to generate better quality results.

Seeding a PRNG

The sequence of "random numbers" generated by a PRNG is not random at all. Just like our plusOne() function, LCG16() is also deterministic. Given some initial state value (such as 0), a PRNG will generate the

same sequence of numbers each time. If you run the above program 3 times, you'll see it generates the same sequence of values each time.
In order to generate different output sequences, the initial state of a PRNG needs to be varied. The value (or set of values) used to set the initial state of a PRNG is called a **random seed** (or seed for short). When the initial state of a PRNG has been set using a seed, we say it has been **seeded**.

Modern PRNG algorithms

Over the years, many different kinds of PRNG algorithms have been developed (Wikipedia has a good list here). Every PRNG algorithm has strengths and weaknesses that might make it more or less suitable for a particular application, so selecting the right algorithm for your application is important.
Many PRNGs are now considered relatively poor by modern standards – and there's no reason to use a PRNG that doesn't perform well when it's just as easy to use one that does.
The randomization capabilities in C++ are accessible via the ¡random¿ header of the standard library. Within the random library, there are 6 PRNG families available for use (as of C++20):
As of C++20, the Mersenne Twister algorithm is the only PRNG that ships with C++ that has both decent performance and quality.

8.9.2 Mersenne Twister

The Mersenne Twister PRNG, besides having a great name, is probably the most popular PRNG across all programming languages. Although it is a bit old by today's standards, it generally produces quality results and has decent performance. The random library has support for two Mersenne Twister types:
`mt19937` is a Mersenne Twister that generates 32-bit unsigned integers `mt19937_64` is a Mersenne Twister that generates 64-bit unsigned integers
Using Mersenne Twister is straightforward:

```
#include <iostream>
#include <random> // for std::mt19937

int main()
{
  std::mt19937 mt{}; // Instantiate a 32-bit Mersenne Twister

  // Print a bunch of random numbers
  for (int count{ 1 }; count <= 40; ++count)
  {
    std::cout << mt() << '\t'; // generate a random number

    // If we've printed 5 numbers, start a new row
    if (count % 5 == 0)
      std::cout << '\n';
  }

  return 0;
}
```

Rolling a dice using Mersenne Twister

The random library has many random numbers distributions, most of which you will never use unless you're doing some kind of statistical analysis. But there's one random number distribution that's extremely useful: a uniform distribution is a random number distribution that produces outputs between two numbers X and Y (inclusive) with equal probability.
Here's a similar program to the one above, using a uniform distribution to simulate the roll of a 6-sided dice:

```
#include <iostream>
#include <random> // for std::mt19937 and std::uniform_int_distribution

int main()
{
  std::mt19937 mt{};

  // Create a reusable random number generator that generates uniform
      numbers between 1 and 6
  std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
      std::uniform_int_distribution<> die6{ 1, 6 };

  // Print a bunch of random numbers
  for (int count{ 1 }; count <= 40; ++count)
  {
    std::cout << die6(mt) << '\t'; // generate a roll of the die here

    // If we've printed 10 numbers, start a new row
    if (count % 10 == 0)
      std::cout << '\n';
  }

  return 0;
}
```

If you run the program multiple times, you will note that it prints the same numbers every time! While each number in the sequence is random with regards to the previous one, the entire sequence is not random at all! Each run of our program produces the exact same result.
Because we are value initializing our Mersenne Twister, it is being initialized with the same seed every time the program is run. And because the seed is the same, the random numbers being generated are also the same.
In order to make our entire sequence randomized differently each time the program is run, we need to pick a seed that's not a fixed number. The first answer that probably comes to mind is that we need a random number for our seed! That's a good thought, but if we need a random number to generate random numbers, then we're in a catch-22. It turns out, we really don't need our seed to be a random number – we just need to pick something that changes each time the program is run. Then we can use our PRNG to generate a unique sequence of pseudo-random numbers from that seed.
There are two methods that are commonly used to do this:

- Use the system clock
- Use the system's random device

Seeding with the system clock

What's one thing that's different every time you run your program? Unless you manage to run your program twice at exactly the same moment in time, the answer is that the current time is different. Therefore, if we use the current time as our seed value, then our program will produce a different set of random numbers each time it is run. C and C++ have a long history of PRNGs being seeded using the current time (using the std::time() function), so you will probably see this in a lot of existing code.
Fortunately, C++ has a chrono library containing various clocks that we can use to generate a seed value. To minimize the chance of two time values being identical if the program is run quickly in succession, we want to use some time measure that changes as quickly as possible. For this, we'll ask the clock how much time has passed since the earliest time it can measure. This time is measured in "ticks", which is a very small unit of time (usually nanoseconds, but could be milliseconds).

```
#include <iostream>
#include <random> // for std::mt19937
#include <chrono> // for std::chrono

int main()
{
  // Seed our Mersenne Twister using steady_clock
  std::mt19937 mt{ static_cast<std::mt19937::result_type>(
    std::chrono::steady_clock::now().time_since_epoch().count()
    ) };

  // Create a reusable random number generator that generates uniform
      numbers between 1 and 6
  std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
      std::uniform_int_distribution<> die6{ 1, 6 };

  // Print a bunch of random numbers
  for (int count{ 1 }; count <= 40; ++count)
  {
    std::cout << die6(mt) << '\t'; // generate a roll of the die here

    // If we've printed 10 numbers, start a new row
    if (count % 10 == 0)
      std::cout << '\n';
  }

  return 0;
}
```

Seeding with the random device

The random library contains a type called `std::random_device` that is an implementation-defined PRNG. Normally we avoid implementation-defined capabilities because they have no guarantees about quality or portability, but this is one of the exception cases. Typically `std::random_device` will ask the OS for a pseudo-random number (how it does this depends on the OS).

```
#include <iostream>
#include <random> // for std::mt19937 and std::random_device

int main()
{
  std::mt19937 mt{ std::random_device{}() };

  // Create a reusable random number generator that generates uniform
      numbers between 1 and 6
  std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
      std::uniform_int_distribution<> die6{ 1, 6 };

  // Print a bunch of random numbers
  for (int count{ 1 }; count <= 40; ++count)
  {
    std::cout << die6(mt) << '\t'; // generate a roll of the die here

    // If we've printed 10 numbers, start a new row
    if (count % 10 == 0)
      std::cout << '\n';
  }

  return 0;
}
```

Many PRNGs can be reseeded after the initial seeding. This essentially re-initializes the state of the random number generator, causing it to generate results starting from the new seed state. Reseeding should generally be avoided unless you have a specific reason to do so, as it can cause the results to be less random, or not random at all.

Only seed a PRNG once

8.9.3 Global random Numbers

What happens if we want to use a random number generator in multiple functions or files? One way is to create (and seed) our PRNG in our main() function, and then pass it everywhere we need it. But that's a lot of passing for something we may only use sporadically, and in many different places. It would add a lot of clutter to our code to pass such an object around.
Alternately, you could create a static local std::mt19937 variable in each function that needs it (static so that it only gets seeded once). However, it's overkill to have every function that uses a random number generator define and seed its own local generator, and the low volume of calls to each generator may lead to lower quality results.
What we really want is a single PRNG object that we can share and access anywhere, across all of our functions and files. The best option here is to create a global random number generator object (inside a namespace!). Remember how we told you to avoid non-const global variables? This is an exception.

Here's a simple, header-only solution that you can include in any code file that needs access to a randomized, self-seeded std::mt19937:
random.h:

```
#ifndef RANDOM_MT_H
#define RANDOM_MT_H

#include <chrono>
#include <random>

// This header-only random namespace implements a self-seeding Mersenne Twister.
// Requires C++17 or newer.
// It can be #included into as many code files as needed (The inline keyword avoids ODR violations)
// Freely redistributable, courtesy of learncpp.com (https://www.learncpp.com/cpp-tutorial/global-random-numbers-random-h/)
namespace random
{
  // Returns a seeded Mersenne Twister
  // Note: we'd prefer to return a std::seed_seq (to initialize a std::mt19937), but std::seed can't be copied, so it can't be returned by value.
  // Instead, we'll create a std::mt19937, seed it, and then return the std::mt19937 (which can be copied).
  inline std::mt19937 generate()
  {
    std::random_device rd{};

    // Create seed_seq with clock and 7 random numbers from std::random_device
    std::seed_seq ss{
      static_cast<std::seed_seq::result_type>(std::chrono::steady_clock::now().time_since_e
        rd(), rd(), rd(), rd(), rd(), rd(), rd() };

    return std::mt19937{ ss };
  }

  // Here's our global std::mt19937 object.
  // The inline keyword means we only have one global instance for our whole program.
  inline std::mt19937 mt{ generate() }; // generates a seeded std::mt19937 and copies it into our global object

  // Generate a random int between [min, max] (inclusive)
      // * also handles cases where the two arguments have different types but can be converted to int
  inline int get(int min, int max)
  {
    return std::uniform_int_distribution{min, max}(mt);
  }

  // The following function templates can be used to generate random numbers in other cases

  // See https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
  // You can ignore these if you don't understand them

  // Generate a random value between [min, max] (inclusive)
  // * min and max must have the same type
  // * return value has same type as min and max
  // * Supported types:
  // *    short, int, long, long long
  // *    unsigned short, unsigned int, unsigned long, or unsigned long long
```

```cpp
  // Sample call: random::get(1L, 6L);             // returns long
  // Sample call: random::get(1u, 6u);             // returns unsigned int
  template <typename T>
  T get(T min, T max)
  {
    return std::uniform_int_distribution<T>{min, max}(mt);
  }

  // Generate a random value between [min, max] (inclusive)
  // * min and max can have different types
        // * return type must be explicitly specified as a template argument
  // * min and max will be converted to the return type
  // Sample call: random::get<std::size_t>(0, 6);  // returns std::size_t
  // Sample call: random::get<std::size_t>(0, 6u); // returns std::size_t
  // Sample call: random::get<std::int>(0, 6u);    // returns int
  template <typename R, typename S, typename T>
  R get(S min, T max)
  {
    return get<R>(static_cast<R>(min), static_cast<R>(max));
  }
}

#endif
```

8.9.4 Using random.h

Generating random numbers using the above is as simple as following these three steps:

- Copy/paste the above code into a file named random.h in your project directory and save it. Optionally add random.h to your project.
- include "random.h" from any .cpp file in your project that needs to generate random numbers.
- Call random::get(min, max) to generate a random number between min and max (inclusive). No initialization or setup is required.

Here is a sample program demonstrating different uses of random.h:
main.cpp

```cpp
#include "random.h" // defines random::mt, random::get(), and random::generate()
#include <cstddef> // for std::size_t
#include <iostream>

int main()
{
  // We can call random::get() to generate random integral values
  // If the two arguments have the same type, the returned value will have that same type.
  std::cout << random::get(1, 6) << '\n';   // returns int between 1 and 6
  std::cout << random::get(1u, 6u) << '\n'; // returns unsigned int between 1 and 6

        // In cases where we have two arguments with different types
        // and/or if we want the return type to be different than the argument types
        // We must specify the return type using a template type argument (between the angled brackets)
  // See https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
  std::cout << random::get<std::size_t>(1, 6u) << '\n'; // returns std::size_t between 1 and 6

  // If we have our own distribution, we can access random::mt directly

  // Let's create a reusable random number generator that generates
      uniform numbers between 1 and 6
  std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
      std::uniform_int_distribution<> die6{ 1, 6 };
  for (int count{ 1 }; count <= 10; ++count)
  {
    std::cout << die6(random::mt) << '\t'; // generate a roll of the die
        here
  }

  std::cout << '\n';

  return 0;
}
```

8.9.5 Simple Random Generation Header

```
#include <chrono>
#include <random>

namespace Random {
inline std::default_random_engine rng(static_cast<size_t>(
    std::chrono::steady_clock::now().time_since_epoch().count()));

// Generate a random integer in range [min, Max]
inline int get(int min, int max) {
  return std ::uniform_int_distribution{min, max}(rng);
}

// Generate a random number in range (min, Max]
template <typename T>
inline T get(T min, T max) {
  return std ::uniform_int_distribution<T>{min, max}(rng);
}

}  // namespace Random
```

Chapter 9

Error detection and handling

9.1 Testing your code

So, you've written a program, it compiles, and it even appears to work! What now? You really should be validating that your program works like you think it does under a wide variety of conditions, and that requires some proactive testing.
Just because your program worked for one set of inputs doesn't mean it's going to work correctly in all cases.

9.1.1 Test your programs in small pieces

Testing a small part of your code in isolation to ensure that "unit" of code is correct is called **unit testing**. Each **unit test** is designed to ensure that a particular behavior of the unit is correct.

9.1.2 Informal testing

One way you can test code is to do informal testing as you write the program. After writing a unit of code (a function, a class, or some other discrete "package" of code), you can write some code to test the unit that was just added, and then erase the test once the test passes. As an example, for the following isLowerVowel() function, you might write the following code:

```cpp
#include <iostream>

// We want to test the following function
// For simplicity, we'll ignore that 'y' is sometimes counted as a vowel
bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    // So here's our temporary tests to validate it works
    std::cout << isLowerVowel('a') << '\n'; // temporary test code, should
        produce 1
    std::cout << isLowerVowel('q') << '\n'; // temporary test code, should
        produce 0

    return 0;
}
```

One problem with the above test function is that it relies on you to manually verify the results when you run it. This requires you to remember what the expected answer was at worst (assuming you didn't document it), and manually compare the actual results to the expected results.
We can do better by writing a test function that contains both the tests AND the expected answers and compares them so we don't have to.

```
#include <iostream>

bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

// returns the number of the test that failed, or 0 if all tests passed
int testVowel()
{
    if (!isLowerVowel('a')) return 1;
    if (isLowerVowel('q')) return 2;

    return 0;
}

int main()
{
    int result { testVowel() };
    if (result != 0)
        std::cout << "testVowel() test " << result << " failed.\n";
    else
        std::cout << "testVowel() tests passed.\n";

    return 0;
}
```

9.1.3 Unit testing frameworks

Because writing functions to exercise other functions is so common and useful, there are entire frameworks (called unit testing frameworks) that are designed to help simplify the process of writing, maintaining, and executing unit tests. Since these involve third party software, we won't cover them here, but you should be aware they exist.
An example of unit testing framework is: Doctest.h

9.1.4 Doctest.h

9.2 Detecting errors

9.3 Handling errors

Most errors in a program don't occur as the result of inadvertently misusing language features – rather, most errors occur due to faulty assumptions made by the programmer and/or a lack of proper error detection/handling.
For example, in a function designed to look up a grade for a student, you might have assumed:

- The student being looked up will exist.

- All student names will be unique.
- The class uses letter grading (instead of pass/fail).

What if any of these assumptions aren't true? If the programmer didn't anticipate these cases, the program will likely malfunction or crash when such cases arise (usually at some point in the future, well after the function has been written).

9.3.1 Handling errors in functions

Functions may fail for any number of reasons – the caller may have passed in an argument with an invalid value, or something may fail within the body of the function. For example, a function that opens a file for reading might fail if the file cannot be found.
When this happens, you have quite a few options at your disposal. There is no best way to handle an error – it really depends on the nature of the problem and whether the problem can be fixed or not.
There are 4 general strategies that can be used:

- Handle the error within the function
- Pass the error back to the caller to deal with
- Halt the program
- Throw an exception

Handling the error within the function

If possible, the best strategy is to recover from the error in the same function in which the error occurred, so that the error can be contained and corrected without impacting any code outside the function. There are two options here: retry until successful, or cancel the operation being executed.
If the error has occurred due to something outside of the program's control, the program can retry until success is achieved. For example, if the program requires an internet connection, and the user has lost their connection, the program may be able to display a warning and then use a loop to periodically recheck for internet connectivity. Alternatively, if the user has entered invalid input, the program can ask the user to try again, and loop until the user is successful at entering valid input. We'll show examples of handling invalid input and using loops to retry in the next lesson.

Passing errors back to the caller

In many cases, the error can't reasonably be handled in the function that detects the error. For example, consider the following function:

```
int doIntDivision(int x, int y)
{
    return x / y;
}
```

In such cases, the best option can be to pass the error back to the caller in hopes that the caller will be able to deal with it.
If the function has a void return type, it can be changed to return a bool that indicates success or failure. If the function returns a normal value, things are a little more complicated.

Fatal errors

If the error is so bad that the program can not continue to operate properly, this is called a non-recoverable error (also called a fatal error). In such cases, the best thing to do is terminate the program. If your code is in main() or a function called directly from main(), the best thing to do is let main() return a non-zero status code. However, if you're deep in some nested subfunction, it may not be convenient or possible to propagate the error all the way back to main(). In such a case, a halt statement (such as std::exit()) can be used.

Exceptions

Because returning an error from a function back to the caller is complicated (and the many different ways to do so leads to inconsistency, and inconsistency leads to mistakes), C++ offers an entirely separate way to pass errors back to the caller: exceptions.
The basic idea is that when an error occurs, an exception is "thrown". If the current function does not "catch" the error, the caller of the function has a chance to catch the error. If the caller does not catch the error, the caller's caller has a chance to catch the error. The error progressively moves up the call stack until it is either caught and handled (at which point execution continues normally), or until main() fails to handle the error (at which point the program is terminated with an exception error).

9.4 Handling invalid input

Most programs that have a user interface of some kind need to handle user input. In the programs that you have been writing, you have been using std::cin to ask the user to enter text input. Because text input is so free-form (the user can enter anything), it's very easy for the user to enter input that is not expected.
Before we discuss how std::cin and operator>> can fail, let's recap how they work.
Here's a simplified view of how operator>> works for input:

- First, leading whitespace (spaces, tabs, and newlines at the front of the buffer) is discarded from the input buffer. This will discard any unextracted newline character remaining from a prior line of input.
- If the input buffer is now empty, operator>> will wait for the user to enter more data. Leading whitespace is again discarded.
- operator>> then extracts as many consecutive characters as it can, until it encounters either a newline character (representing the end of the line of input) or a character that is not valid for the variable being extracted to.

The result of the extraction is as follows:

- If any characters were extracted in step 3 above, extraction is a success. The extracted characters are converted into a value that is then assigned to the variable.
- If no characters could be extracted in step 3 above, extraction has failed. The object being extracted to is assigned the value 0 (as of C++11), and any future extractions will immediately fail (until std::cin is cleared).

Validating input

The process of checking whether user input conforms to what the program is expecting is called **input validation**.
There are three basic ways to do input validation:
Inline (as the user types):
1. Prevent the user from typing invalid input in the first place.
Post-entry (after the user types):
2. Let the user enter whatever they want into a string, then validate whether the string is correct, and if so, convert the string to the final variable format.
3. Let the user enter whatever they want, let std::cin and operator>> try to extract it, and handle the error cases.
Some graphical user interfaces and advanced text interfaces will let you validate input as the user enters it (character by character). Generally speaking, the programmer provides a validation function that accepts the input the user has entered so far, and returns true if the input is valid, and false otherwise. This function is called every time the user presses a key. If the validation function returns true, the key the user just pressed is accepted. If the validation function returns false, the character the user just input is discarded (and not shown on the screen). Using this method, you can ensure that any input the user enters is guaranteed to be valid, because any invalid keystrokes are discovered and discarded immediately. Unfortunately, std::cin does not support this style of validation.

Types of invalid text input

We can generally separate input text errors into four types:

- Input extraction succeeds but the input is meaningless to the program (e.g. entering 'k' as your mathematical operator).
- Input extraction succeeds but the user enters additional input (e.g. entering '*q hello' as your mathematical operator).
- Input extraction fails (e.g. trying to enter 'q' into a numeric input).
- Input extraction succeeds but the user overflows a numeric value.

Handling std::cin

There are various possibilities for handling std::cin input, but when creating a project is preferrable to do it with the usage of a graphic interface which allows to do that in a simpler and cleaner way.
Anyways, here's a simple program that allows to do some checks on the stf::cin input

```cpp
#include <cstdlib> // for std::exit
#include <iostream>
#include <limits> // for std::numeric_limits

void ignoreLine()
{
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

// returns true if extraction failed, false otherwise
bool clearFailedExtraction()
{
    // Check for failed extraction
    if (!std::cin) // If the previous extraction failed
    {
        if (std::cin.eof()) // If the stream was closed
        {
            std::exit(0); // Shut down the program now
        }

        // Let's handle the failure
        std::cin.clear(); // Put us back in 'normal' operation mode
        ignoreLine();     // And remove the bad input

        return true;
    }

    return false;
}

double getDouble()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter a decimal number: ";
        double x{};
        std::cin >> x;

        if (clearFailedExtraction())
        {
            std::cout << "Oops, that input is invalid.  Please try
                again.\n";
            continue;
        }

        ignoreLine(); // Remove any extraneous input
        return x;     // Return the value we extracted
    }
}

char getOperator()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char operation{};
        std::cin >> operation;

        if (!clearFailedExtraction()) // we'll handle error messaging if
            extraction failed below
            ignoreLine(); // remove any extraneous input (only if
                extraction succeded)

        // Check whether the user entered meaningful input
        switch (operation)
        {
        case '+':
        case '-':
        case '*':
        case '/':
            return operation; // Return the entered char to the caller
        default: // Otherwise tell the user what went wrong
            std::cout << "Oops, that input is invalid.  Please try
                again.\n";
        }
    }
}

void printResult(double x, char operation, double y)
{
    std::cout << x << ' ' << operation << ' ' << y << " is ";

    switch (operation)
    {
    case '+':
        std::cout << x + y << '\n';
        return;
    case '-':
        std::cout << x - y << '\n';
        return;
    case '*':
        std::cout << x * y << '\n';
        return;
    case '/':
        if (y == 0.0)
            break;

        std::cout << x / y << '\n';
        return;
    }

    std::cout << "???";  // Being robust means handling unexpected
        parameters as well, even though getOperator() guarantees operation
        is valid in this particular program
}

int main()
{
    double x{ getDouble() };
    char operation{ getOperator() };
    double y{ getDouble() };

    // Handle division by 0
```

```cpp
    while (operation == '/' && y == 0.0)
    {
        std::cout << "The denominator cannot be zero.  Try again.\n";
        y = getDouble();
    }

    printResult(x, operation, y);

    return 0;
}
```

9.5 Assert and `static_assert`

9.5.1 Preconditions, invariants and postconditions

In programming, a **precondition** is any condition that must be true prior to the execution of some section of code.
Preconditions for a function are best placed at the top of a function, using an early return to return back to the caller if the precondition isn't met. For example:

```cpp
void printDivision(int x, int y)
{
    if (y == 0) // handle
    {
        std::cerr << "Error: Could not divide by zero\n";
        return; // bounce the user back to the caller
    }

    // We now know that y != 0
    std::cout << static_cast<double>(x) / y;
}
```

An **invariant** is a condition that must be true while some section of code is executing. This is often used with loops, where the loop body will only execute so long as the invariant is true.
Similarly, a **postcondition** is something that must be true after the execution of some section of code. Our function doesn't have any postconditions.

9.5.2 Assertions

An **assertion** is an expression that will be true unless there is a bug in the program. If the expression evaluates to true, the assertion statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated (via std::abort). This error message typically contains the expression that failed as text, along with the name of the code file and the line number of the assertion. This makes it very easy to tell not only what the problem was, but where in the code the problem occurred. This can help with debugging efforts immensely.
In C++, runtime assertions are implemented via the assert preprocessor macro, which lives in the ¡cassert¿ header.

```cpp
#include <cassert> // for assert()
#include <cmath> // for std::sqrt
#include <iostream>

double calculateTimeUntilObjectHitsGround(double initialHeight, double
    gravity)
{
  assert(gravity > 0.0); // The object won't reach the ground unless there
      is positive gravity.

  if (initialHeight <= 0.0)
  {
    // The object is already on the ground. Or buried.
    return 0.0;
  }

  return std::sqrt((2.0 * initialHeight) / gravity);
}

int main()
{
  std::cout << "Took " << calculateTimeUntilObjectHitsGround(100.0, -9.8)
      << " second(s)\n";

  return 0;
}
```

Note: Make your assert statements more descriptive
Fortunately, there's a little trick you can use to make your assert statements more descriptive. Simply add a string literal joined by a logical AND:

```
assert(found && "Car could not be found in database");
```

Here's why this works: A string literal always evaluates to Boolean true. So if found is false, false && true is false. If found is true, true && true is true. Thus, logical AND-ing a string literal doesn't impact the evaluation of the assert.
However, when the assert triggers, the string literal will be included in the assert message:

```
Assertion failed: found && "Car could not be found in database", file
    C:\\VCProjects\\Test.cpp, line 34
```

Note: Assertions are also sometimes used to document cases that were not implemented because they were not needed at the time the programmer wrote the code.
The assert macro comes with a small performance cost that is incurred each time the assert condition is checked. Furthermore, asserts should (ideally) never be encountered in production code (because your code should already be thoroughly tested). Consequently, most developers prefer that asserts are only active in debug builds. C++ comes with a built-in way to turn off asserts in production code: if the preprocessor macro NDEBUG is defined, the assert macro gets disabled.

9.5.3 static_assert

C++ also has another type of assert called static_assert. A static_assert is an assertion that is checked at compile-time rather than at runtime, with a failing static_assert causing a compile error. Unlike assert, which is declared in the ¡cassert¿ header, static_assert is a keyword, so no header needs to be included to use it.
A static_assert takes the following form:

```
static_assert(condition, diagnostic_message)
```

If the condition is not true, the diagnostic message is printed. Here's an example of using static_assert to ensure types have a certain size:

```
static_assert(sizeof(long) == 8, "long must be 8 bytes");
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");

int main()
{
  return 0;
}
```

A few useful notes about static_assert:

- Because static_assert is evaluated by the compiler, the condition must be a constant expression.
- static_assert can be placed anywhere in the code file (even in the global namespace).
- static_assert is not deactivated in release builds (like normal assert is).
- Because the compiler does the evaluation, there is no runtime cost to a static_assert.

9.5.4 Asserts vs error handling

Assertions and error handling are similar enough that their purposes can be confused, so let's clarify.
Assertions are used to detect programming errors during development by documenting assumptions about things that should never happen. And if they do happen, it's the fault of the programmer. Assertions do not allow recovery from errors (after all, if something should never happen, there's no need to recover from it). Because asserts are typically compiled-out in release builds, you can put a lot of them in without worrying about performance, so there's little reason not to use them liberally.
Error handling is used when we need to gracefully handle cases that could happen (however rarely) in a release build. These may either be recoverable issues (where the program can keep running), or unrecoverable issues (where the program has to shut down, but we can at least show a nice error message and ensure everything is cleaned up properly). Error detection and handling has both a runtime performance cost and a development time cost.

Chapter 10

Templates

10.1 Function Overloading

Function overloading allows us to create multiple functions with the same name, so long as each identically named function has different parameter types (or the functions can be otherwise differentiated). Each function sharing a name (in the same scope) is called an **overloaded function** (sometimes called an overload for short).

```
int add(int x, int y) // integer version
{
    return x + y;
}

double add(double x, double y) // floating point version
{
    return x + y;
}
```

The above program will compile. Although you might expect these functions to result in a naming conflict, that is not the case here. Because the parameter types of these functions differ, the compiler is able to differentiate these functions, and will treat them as separate functions that just happen to share a name.

10.1.1 Overload resolution

Additionally, when a function call is made to a function that has been overloaded, the compiler will try to match the function call to the appropriate overload based on the arguments used in the function call. This is called **overload resolution**.
Here's a simple example demonstrating this:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

int main()
{
    std::cout << add(1, 2); // calls add(int, int)
    std::cout << '\n';
    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;
}
```

In order for a program using overloaded functions to compile, two things have to be true:

- Each overloaded function has to be differentiated from the others. We discuss how functions can be differentiated in lesson 11.2 – Function overload differentiation.
- Each call to an overloaded function has to resolve to an overloaded function. We discuss how the compiler matches function calls to overloaded functions in lesson 11.3 – Function overload resolution and ambiguous matches.

If an overloaded function is not differentiated, or if a function call to an overloaded function can not be resolved to an overloaded function, then a compile error will result.

10.1.2 How overloaded functions are differentiated

Function property	Used for differentiation	Notes
Number of parameters	Yes	
Type of parameters	Yes	Excludes typedefs, type aliases, and const qualifier on value parameters. Includes ellipses.
Return type	No	

Example of overloading based on number of parameters:

```
int add(int x, int y)
{
    return x + y;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Note: The return type of a function is not considered for differentiation

```
int getRandomValue();
double getRandomValue();
```

On Visual Studio 2019, this results in the following compiler error:

```
error C2556: 'double getRandomValue(void)': overloaded function differs
    only by return type from 'int getRandomValue(void)'
```

10.1.3 Overload resolution

With overloaded functions, there can be many functions that can potentially match a function call. Since a function call can only resolve to one of them, the compiler has to determine which overloaded function is the best match. The process of matching function calls to a specific overloaded function is called overload resolution.
In simple cases where the type of the function arguments and type of the function parameters match exactly, this is (usually) straightforward:

```
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

void print(double d)
{
    std::cout << d << '\n';
}

int main()
{
    print(5); // 5 is an int, so this matches print(int)
```

```cpp
    print(6.7); // 6.7 is a double, so this matches print(double)

    return 0;
}
```

But what happens in cases where the argument types in the function call don't exactly match the parameter types in any of the overloaded functions? For example:

```cpp
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

void print(double d)
{
    std::cout << d << '\n';
}

int main()
{
    print('a'); // char does not match int or double, so what happens?
    print(5L); // long does not match int or double, so what happens?

    return 0;
```

Just because there is no exact match here doesn't mean a match can't be found – after all, a char or long can be implicitly type converted to an int or a double. But which is the best conversion to make in each case?
When a function call is made to an overloaded function, the compiler steps through a sequence of rules to determine which (if any) of the overloaded functions is the best match (we'll cover these steps in the next section below).
At each step, the compiler applies a bunch of different type conversions to the argument(s) in the function call. For each conversion applied, the compiler checks if any of the overloaded functions are now a match. After all the different type conversions have been applied and checked for matches, the step is done. The result will be one of three possible outcomes:

- No matching functions were found. The compiler moves to the next step in the sequence.
- A single matching function was found. This function is considered to be the best match. The matching process is now complete, and subsequent steps are not executed.
- More than one matching function was found. The compiler will issue an ambiguous match compile error. We'll discuss this case further in a bit.

If the compiler reaches the end of the entire sequence without finding a match, it will generate a compile error that no matching overloaded function could be found for the function call.

10.1.4 The argument matching sequence

Step 1) The compiler tries to find an exact match. This happens in two phases. First, the compiler will see if there is an overloaded function where the type of the arguments in the function call exactly matches the type of the parameters in the overloaded functions.

Trivial Conversions

Second, the compiler will apply a number of trivial conversions to the arguments in the function call. The trivial conversions are a set of specific conversion rules that will modify types (without modifying the value) for purposes of finding a match. These include:

- lvalue to rvalue conversions
- qualification conversions (e.g. non-const to const)
- non-reference to reference conversions

For example:

```cpp
void foo(const int)
{
}

void foo(const double&) // double& is a reference to a double
{
}

int main()
{
    int x { 1 };
    foo(x); // x trivially converted from int to const int

    double d { 2.3 };
    foo(d); // d trivially converted from double to const double& (non-ref
        to ref conversion)

    return 0;
}
```

Step 2) If no exact match is found, the compiler tries to find a match by applying numeric promotion to the argument(s). We already covered how certain narrow integral and floating point types can be automatically promoted to wider types, such as int or double. If, after numeric promotion, a match is found, the function call is resolved.
Step 3) If no match is found via numeric promotion, the compiler tries to find a match by applying numeric conversions (10.3 – Numeric conversions) to the arguments.
For example:

```cpp
#include <string> // for std::string

void foo(double)
{
}

void foo(std::string)
{
}

int main()
{
    foo('a'); // 'a' converted to match foo(double)

    return 0;
}
```

Step 4) If no match is found via numeric conversion, the compiler tries to find a match through any user-defined conversions. Although we haven't covered user-defined conversions yet, certain types (e.g. classes) can define conversions to other types that can be implicitly invoked.
Step 5) If no match is found via user-defined conversion, the compiler will look for a matching function that uses ellipsis.
Step 6) If no matches have been found by this point, the compiler gives up and will issue a compile error about not being able to find a matching function.

Ambiguous matches

With non-overloaded functions, each function call will either resolve to a function, or no match will be found and the compiler will issue a compile error.
With overloaded functions, there is a third possible outcome: an ambiguous match may be found. An ambiguous match occurs when the compiler finds two or more functions that can be made to match in the same step. When this occurs, the compiler will stop matching and issue a compile error stating that it has found an ambiguous function call.
Since every overloaded function must be differentiated in order to compile, you might be wondering how it is possible that a function call could result in more than one match. Let's take a look at an example that illustrates this:

```cpp
void foo(int)
{
}

void foo(double)
{
}

int main()
{
    foo(5L); // 5L is type long

    return 0;
}
```

Since literal 5L is of type long, the compiler will first look to see if it can find an exact match for foo(long), but it will not find one. Next, the compiler will try numeric promotion, but values of type long can't be promoted, so there is no match here either.
Following that, the compiler will try to find a match by applying numeric conversions to the long argument. In the process of checking all the numeric conversion rules, the compiler will find two potential matches. If the long argument is numerically converted into an int, then the function call will match foo(int). If the long argument is instead converted into a double, then it will match foo(double) instead. Since two possible matches via numeric conversion have been found, the function call is considered ambiguous.

10.2 Deleting functions

In cases where we have a function that we explicitly do not want to be callable, we can define that function as deleted by using the = delete specifier. If the compiler matches a function call to a deleted function, compilation will be halted with a compile error.

```cpp
#include <iostream>

void printInt(int x)
{
    std::cout << x << '\n';
}

void printInt(char) = delete; // calls to this function will halt
    compilation
void printInt(bool) = delete; // calls to this function will halt
    compilation

int main()
{
    printInt(97);   // okay

    printInt('a');  // compile error: function deleted
    printInt(true); // compile error: function deleted

    printInt(5.0);  // compile error: ambiguous match

    return 0;
}
```

10.2.1 Deleting all non-matching overloads

Deleting a bunch of individual function overloads works fine, but can be verbose. There may be times when we want a certain function to be called only with arguments whose types exactly match the function parameters. We can do this by using a function template (which we will be talking about next):

```cpp
#include <iostream>

// This function will take precedence for arguments of type int
```

```cpp
void printInt(int x)
{
    std::cout << x << '\n';
}

// This function template will take precedence for arguments of other types
// Since this function template is deleted, calls to it will halt
    compilation
template <typename T>
void printInt(T x) = delete;

int main()
{
    printInt(97);   // okay
    printInt('a');  // compile error
    printInt(true); // compile error

    return 0;
}
```

10.3 Default arguments

A **default argument** is a default value provided for a function parameter. For example:

```cpp
void print(int x, int y=10) // 10 is the default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

When making a function call, the caller can optionally provide an argument for any function parameter that has a default argument. If the caller provides an argument, the value of the argument in the function call is used. If the caller does not provide an argument, the value of the default argument is used.

```cpp
#include <iostream>

void print(int x, int y=4) // 4 is the default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    print(1, 2); // y will use user-supplied argument 2
    print(3); // y will use default argument 4, as if we had called
        print(3, 4)

    return 0;
}
```

10.3.1 When to use default arguments

Default arguments are an excellent option when a function needs a value that has a reasonable default value, but for which you want to let the caller override if they wish.
For example, here are a couple of function prototypes for which default arguments might be commonly used:

```cpp
int rollDie(int sides=6);
void openLogFile(std::string filename="default.log");
```

10.3.2 Multiple default arguments

A function can have multiple parameters with default arguments:

```cpp
#include <iostream>

void print(int x=10, int y=20, int z=30)
{
    std::cout << "Values: " << x << " " << y << " " << z << '\n';
}

int main()
{
    print(1, 2, 3); // all explicit arguments
    print(1, 2); // rightmost argument defaulted
    print(1); // two rightmost arguments defaulted
    print(); // all arguments defaulted

    return 0;
}
```

C++ does not (as of C++23) support a function call syntax such as print(,,3) (as a way to provide an explicit value for z while using the default arguments for x and y. This has three major consequences:

- In a function call, any explicitly provided arguments must be the leftmost arguments (arguments with defaults cannot be skipped).
- If a parameter is given a default argument, all subsequent parameters (to the right) must also be given default arguments.
- If more than one parameter has a default argument, the leftmost parameter should be the one most likely to be explicitly set by the user.

```cpp
void print(int x=10, int y); // not allowed
```

10.3.3 Declaration and definition

Default arguments can not be redeclared, and must be declared before use:

```cpp
#include <iostream>

void print(int x, int y=4); // forward declaration

void print(int x, int y=4) // compile error: redefinition of default
    argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

The default argument must also be declared in the translation unit before it can be used.
The best practice is to declare the default argument in the forward declaration and not in the function definition, as the forward declaration is more likely to be seen by other files and included before use (particularly if it's in a header file).
foo.h

```cpp
#ifndef FOO_H
#define FOO_H
void print(int x, int y=4);
#endif
```

main.cpp

```cpp
#include "foo.h"
#include <iostream>

void print(int x, int y)
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    print(5);

    return 0;
}
```

10.3.4 Default arguments and function overloading

Functions with default arguments may be overloaded. For example, the following is allowed:

```
void print(char c = ' ')
{
    std::cout << c << '\n';
}
```

Note: This can lead to ambiguous matches and should be used carefully, consider:

```
void print(int x);                    // signature print(int)
void print(int x, int y = 10);        // signature print(int, int)
void print(int x, double y = 20.5); // signature print(int, double)

int main()
{
    print(1, 2);    // will resolve to print(int, int)
    print(1, 2.5); // will resolve to print(int, double)
    print(1);       // ambiguous function call

    return 0;
}
```

For the call print(1), the compiler is unable to tell whether this resolve to print(int), print(int, int), or print(int, double).

10.4 Function templates

10.4.1 Introduction to function templates

Let's say you wanted to write a function to calculate the maximum of two numbers. You might do so like this:

```
int max(int x, int y)
{
    return (x < y) ? y : x;
    // Note: we use < instead of > because std::max uses <
}
```

While the caller can pass different values into the function, the type of the parameters is fixed, so the caller can only pass in int values. That means this function really only works well for integers (and types that can be promoted to int).
In C++, the template system was designed to simplify the process of creating functions (or classes) that are able to work with different data types.
Instead of manually creating a bunch of mostly-identical functions or classes (one for each set of different types), we instead create a single **template**. Just like a normal definition, a template definition describes what a function or class looks like. Unlike a normal definition (where all types must be specified), in a template we can use one or more placeholder types. A placeholder type represents some type that is not known at the time the template is defined, but that will be provided later (when the template is used).
Once a template is defined, the compiler can use the template to generate as many overloaded functions (or classes) as needed, each using different actual types!
The end result is the same – we end up with a bunch of mostly-identical functions or classes (one for each set of different types). But we only have to create and maintain a single template, and the compiler does all the hard work to create the rest for us.

10.4.2 Definition

A **function template** is a function-like definition that is used to generate one or more overloaded functions, each with a different set of actual types. This is what will allow us to create functions that can work with many different types. The initial function template that is used to generate other functions is called the primary template, and the functions generated from the primary template are called instantiated functions.
When we create a primary function template, we use placeholder types (technically called type template parameters, informally called template types) for any parameter types, return types, or types used in the function body that we want to be specified later, by the user of the template.
Here's our new function that uses a single template type, where all occurrences of actual type int have been replaced with type template parameter T, we're going to tell the compiler that this is a template, and that T is a type template parameter that is a placeholder for any type. Both of these are done using a template parameter declaration, which defines any template parameters that will be subsequently used. The scope of a template parameter declaration is strictly limited to the function template (or class template) that follows. Therefore, each function template or class template needs its own template parameter declaration.

```
template <typename T> // this is the template parameter declaration
    defining T as a type template parameter
T max(T x, T y) // this is the function template definition for max<T>
{
    return (x < y) ? y : x;
}
```

In our template parameter declaration, we start with the keyword template, which tells the compiler that we're creating a template. Next, we specify all of the template parameters that our template will use inside angled brackets (¡¿). For each type template parameter, we use the keyword typename (preferred) or class, followed by the name of the type template parameter (e.g. T).

10.4.3 Naming conventions

Much like we often use a single letter for variable names used in trivial situations (e.g. x), it's conventional to use a single capital letter (starting with T) when the template parameter is used in a trivial or obvious way.
We don't need to give T a complex name, because it's obviously just a placeholder type for the values being compared, and T can be any type that can be compared (such as int, double, or char, but not nullptr).
Our function templates will generally use this naming convention.
If a type template parameter has a non-obvious usage or specific requirements that must be met, there are two common conventions for such names:

- Starting with a capital letter (e.g. Allocator). The standard library uses this naming convention.
- Prefixed with a T, then starting with a capital letter (e.g. TAllocator). This makes it easier to see that the type is a type template parameter.

10.4.4 Using templates

This looks a lot like a normal function call – the primary difference is the addition of the type in angled brackets (called a template argument), which specifies the actual type that will be used in place of template type T.

```
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls
        function max<int>(int, int)

    return 0;
}
```

When the compiler encounters the function call max¡int¿(1, 2), it will determine that a function definition for max¡int¿(int, int) does not already exist. Consequently, the compiler will implicitly use our max¡T¿ function template to create one.
The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation** (or instantiation for short). When a function is instantiated due to a function call, it's called implicit instantiation. A function that is instantiated from a template is technically called a specialization, but in common language is often called a function instance. The template from which a specialization is produced is called a primary template. Function instances are normal functions in all regards.

10.4.5 Template argument deduction

In cases where the type of the arguments match the actual type we want, we do not need to specify the actual type – instead, we can use template argument deduction to have the compiler deduce the actual type that should be used from the argument types in the function call.

```cpp
std::cout << max<>(1, 2) << '\n';
std::cout << max(1, 2) << '\n';
```

10.4.6 Function templates with non-template parameters

It's possible to create function templates that have both template parameters and non-template parameters. The type template parameters can be matched to any type, and the non-template parameters work like the parameters of normal functions.

```cpp
// T is a type template parameter
// double is a non-template parameter
// We don't need to provide names for these parameters since they aren't
    used
template <typename T>
int someFcn(T, double)
{
    return 5;
}

int main()
{
    someFcn(1, 3.4); // matches someFcn(int, double)
    someFcn(1, 3.4f); // matches someFcn(int, double) -- the float is
        promoted to a double
    someFcn(1.2, 3.4); // matches someFcn(double, double)
    someFcn(1.2f, 3.4); // matches someFcn(float, double)
    someFcn(1.2f, 3.4f); // matches someFcn(float, double) -- the float is
        promoted to a double

    return 0;
}
```

10.4.7 Multiple template types

Rather than using one template type parameter T, we'll now use two (T and U):

```cpp
#include <iostream>

template <typename T, typename U> // We're using two template type
    parameters named T and U
T max(T x, U y) // x can resolve to type T, and y can resolve to type U
{
    return (x < y) ? y : x; // uh oh, we have a narrowing conversion
        problem here
}

int main()
{
    std::cout << max(2, 3.5) << '\n'; // resolves to max<int, double>

    return 0;
}
```

Because we've defined x with template type T, and y with template type U, x and y can now resolve their types independently. When we call max(2, 3.5), T can be an int and U can be a double. The compiler will happily instantiate max¡int, double¿(int, double) for us.
However, this example doesn't work right. If you compile and run the program (with "treat warnings as errors" turned off), it will produce the following result:

```
3
```

The conditional operator (?:) requires its (non-condition) operands to be the same common type. The usual arithmetic rules (10.5 – Arithmetic conversions) are used to determine what the common type will be, and the result of the conditional operator will also use this common type. For example, the common type of int and double is double, so when the (non-condition) operands of our conditional operator are an int and a double, the value produced by the conditional operator will be of type double. In this case, that's the value 3.5, which is correct.
However, the declared return type of our function is T. When T is an int and U is a double, the return type of the function is int. Our value 3.5 is undergoing a narrowing conversion to int value 3, resulting in a loss of data (and possibly a compiler warning).
So how do we solve this? Making the return type a U instead doesn't solve the problem, as max(3.5, 2) has U as an int and will exhibit the same issue.
In such cases, return type deduction (via auto) can be useful – we'll let the compiler deduce what the return type should be from the return statement:

```
#include <iostream>

template <typename T, typename U>
auto max(T x, U y) // ask compiler can figure out what the relevant return
    type is
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

This version of max now works fine with operands of different types. Just note that a function with an auto return type needs to be fully defined before it can be used (a forward declaration won't suffice), since the compiler has to inspect the function implementation to determine the return type.

10.4.8 Function templates may be overloaded

Just like functions may be overloaded, function templates may also be overloaded. Such overloads can have a different number of template types and/or a different number or type of function parameters:

```
#include <iostream>

// Add two values with matching types
template <typename T>
auto add(T x, T y)
{
    return x + y;
}

// Add two values with non-matching types
// As of C++20 we could also use auto add(auto x, auto y)
template <typename T, typename U>
auto add(T x, U y)
{
    return x + y;
}

// Add three values with any type
// As of C++20 we could also use auto add(auto x, auto y, auto z)
template <typename T, typename U, typename V>
auto add(T x, U y, V z)
{
    return x + y + z;
}

int main()
{
    std::cout << add(1.2, 3.4) << '\n'; // instantiates and calls
        add<double>()
    std::cout << add(5.6, 7) << '\n';   // instantiates and calls
        add<double, int>()
    std::cout << add(8, 9, 10) << '\n'; // instantiates and calls add<int,
        int, int>()

    return 0;
}
```

One interesting note here is that for the call to add(1.2, 3.4), the compiler will prefer add¡T¿(T, T) over add¡T, U¿(T, U) even though both could possibly match.
The rules for determining which of multiple matching function templates should be preferred are called "partial ordering of function templates". In short, whichever function template is more restrictive/specialized will be preferred. add¡T¿(T, T) is the more restrictive function template in this case (since it only has one template parameter), so it is preferred.
If multiple function templates can match a call and the compiler can't determine which is more restrictive, the compiler will error with an ambiguous match.

10.5 Abbreviated function templates

C++20 introduces a new use of the auto keyword: When the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an **abbreviated function template**.

```
auto max(auto x, auto y)
{
    return (x < y) ? y : x;
}
```

However, there isn't a concise way to use abbreviated function templates when you want more than one auto parameter to be the same type. That is, there isn't an easy abbreviated function template for something like this:

```
template <typename T>
T max(T x, T y) // two parameters of the same type
{
    return (x < y) ? y : x;
}
```

Best practice: Feel free to use abbreviated function templates with a single auto parameter, or where each auto parameter should be an independent type (and your language standard is set to C++20 or newer).

10.6 Non-type template parameters

10.7 Function templates in multiple files

Consider the following program, which doesn't work correctly:

main.cpp

```cpp
#include <iostream>

template <typename T>
T addOne(T x); // function template forward declaration

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

add.cpp

```cpp
template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}
```

If addOne were a non-template function, this program would work fine: In main.cpp, the compiler would be satisfied with the forward declaration of addOne, and the linker would connect the call to addOne() in main.cpp to the function definition in add.cpp.
The most conventional way to address this issue is to put all your template code in a header (.h) file instead of a source (.cpp) file.

10.7.1 Correct implementation

add.h:

```cpp
#ifndef ADD_H
#define ADD_H

template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}

#endif
```

main.cpp

```cpp
#include "add.h" // import the function template definition
#include <iostream>

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

That way, any files that need access to the template can include the relevant header, and the template definition will be copied by the preprocessor into the source file. The compiler will then be able to instantiate any functions that are needed.
You may be wondering why this doesn't cause a violation of the one-definition rule (ODR). The ODR says that types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. So there is no problem if the template definition is copied into multiple files (as long as each definition is identical).

Chapter 11

References

11.1 Introduction to compund data types

Imagine you were writing a math program to multiply two fractions. How would you represent a fraction in your program? You might use a pair of integers (one for the numerator, one for the denominator), like this:

```cpp
#include <iostream>

int main()
{
    // Our first fraction
    int num1 {};
    int den1 {};

    // Our second fraction
    int num2 {};
    int den2 {};

    // Used to eat (remove) the slash between the numerator and denominator
    char ignore {};

    std::cout << "Enter a fraction: ";
    std::cin >> num1 >> ignore >> den1;

    std::cout << "Enter a fraction: ";
    std::cin >> num2 >> ignore >> den2;

    std::cout << "The two fractions multiplied: "
        << num1 * num2 << '/' << den1 * den2 << '\n';

    return 0;
}
```

While this program works, it introduces a couple of challenges for us to improve upon. First, each pair of integers is only loosely linked – outside of comments and the context of how they are used in the code, there's little to suggest that each numerator and denominator pair are related. Second, following the DRY (don't repeat yourself) principle, we should create a function to handle the user inputting a fraction (along with some error handling). However, functions can only return a single value, so how would we return the numerator and denominator back to the caller?

Now imagine another case where you're writing a program that needs to keep a list of employee IDs. How might you do so? But what if you had 100 employees? First, you'd need to type in 100 variable names. And what if you needed to print them all? Or pass them to a function? We'd be in for a lot of typing. This simply doesn't scale.

Fortunately, C++ supports a second set of data types: **compound data types** (also sometimes called composite data types) are types that are defined in terms of other existing data types. Compound data types have additional properties and behaviors that make them useful for solving certain types of problems.

As we'll show in this chapter and future chapters, we can use compound data types to elegantly solve all of the challenges we presented above.

C++ supports the following compound types:

- Functions
- C-style Arrays
- Pointer types:
  - Pointer to object
  - Pointer to function
- Pointer to member types:
  - Pointer to data member
  - Pointer to member function
- Reference types:
  - L-value references
  - R-value references
- Enumerated types:
  - Unscoped enumerations
  - Scoped enumerations
- Class types:
  - Structs
  - Classes
  - Unions

You've already been using one compound type regularly: functions. For example, consider this function:

```cpp
void doSomething(int x, double y)
{
}
```

The type of this function is void(int, double). Note that this type is composed of fundamental types, making it a compound type. Of course, functions also have their own special behaviors as well (e.g. being callable).

11.1.1 Value categories (lvalues and rvalues)

https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/

```cpp
    x = 5; // valid: we can assign 5 to x
    5 = x; // error: can not assign value of x to literal value 5
```

The value category of an expression (or subexpression) indicates whether an expression resolves to a value, a function, or an object of some kind.

Prior to C++11, there were only two possible value categories: **lvalue** and **rvalue**. In C++11, three additional value categories (glvalue, prvalue, and xvalue) were added to support a new feature called move semantics.

lvalue : Expression that evaluates to an identifiable object or function (or bit-field).

```cpp
    int x { 5 };
```

Entities with identities can be accessed via an identifier, reference, or pointer, and typically have a lifetime longer than a single expression or statement.

lvalues come in two subtypes: a **modifiable lvalue** is an lvalue whose value can be modified. A **non-modifiable lvalue** is an lvalue whose value can't be modified.

```cpp
    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
```

rvalue : Expression that is not an lvalue. Rvalue expressions evaluate to a value. Commonly seen rvalues include literals (except C-style string literals, which are lvalues) and the return value of functions and operators that return by value. Rvalues aren't identifiable and only exist within the scope of the expression in which they are used.

```cpp
int return5()
{
    return 5;
}

int main()
{
    int x{ 5 }; // 5 is an rvalue expression
    const double d{ 1.2 }; // 1.2 is an rvalue expression

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
    int z { return5() }; // return5() is an rvalue expression (since the
        result is returned by value)

    int w { x + 1 }; // x + 1 is an rvalue expression
    int q { static_cast<int>(d) }; // the result of static casting d to an
        int is an rvalue expression

    return 0;
}
```

You may be wondering why return5(), $x+1$, and static_cast are rvalues: the answer is because these expressions produce temporary values that are not identifiable objects.
Now we can answer the question about why x = 5 is valid but 5 = x is not: an assignment operation requires the left operand of the assignment to be a modifiable lvalue expression, and the right operand to be an rvalue expression.

Lvalue to rvalue conversion:

```cpp
    int x { 5 };
    int y { x }; // x is an lvalue expression
```

lvalue expressions will implicitly convert to rvalue expressions in contexts where an rvalue is expected but an lvalue is provided.

Lvalue-to-rvalue conversion

```cpp
int main()
{
    int x{ 1 };
    int y{ 2 };

    x = y; // y is not an rvalue, but this is legal

    return 0;
}
```

In cases where an rvalue is expected but an lvalue is provided, the lvalue will undergo an lvalue-to-rvalue conversion so that it can be used in such contexts. This basically means the lvalue is evaluated to produce its value, which is an rvalue.
Now consider this example:

```cpp
int main()
{
    int x { 2 };

    x = x + 1;

    return 0;
}
```

In this statement, the variable x is being used in two different contexts. On the left side of the assignment operator (where an lvalue expression is required), x is an lvalue expression that evaluates to variable x. On the right side of the assignment operator, x undergoes an lvalue-to-rvalue conversion and is then evaluated so that its (2) can be used as the left operand of operator+. operator+ returns the rvalue expression 3, which is then used as the right operand for the assignment.

11.2 Lvalue references

`https://www.learncpp.com/cpp-tutorial/lvalue-references/`
In C++, a **reference** is an alias for an existing object. Once a reference has been defined, any operation on the reference is applied to the object being referenced. This means we can use a reference to read or modify the object being referenced.
Just like the type of an object determines what kind of value it can hold, the type of a reference determines what type of object it can reference. Lvalue reference types can be identified by use of a single ampersand (&) in the type specifier:

```
int&        // an lvalue reference to an int object
double&     // an lvalue reference to a double object
const int& // an lvalue reference to a const int object
```

A type that specifies a reference (e.g. int&) is called a reference type. The type that can be referenced (e.g. int) is called the referenced type.

11.2.1 Reference initialization

Much like constants, all references must be initialized. References are initialized using a form of initialization called reference initialization.

```
    int& invalidRef;   // error: references must be initialized

    int x { 5 };
    int& ref { x }; // okay: reference to int is bound to int variable
```

When a reference is initialized with an object (or function), we say it is bound to that object (or function). The process by which such a reference is bound is called reference binding. The object (or function) being referenced is sometimes called the referent.

Non-const lvalue references can only be bound to a modifiable lvalue.

```
    int x { 5 };
    int& ref { x };          // okay: non-const lvalue reference bound to a
        modifiable lvalue

    const int y { 5 };
    int& invalidRef { y };  // invalid: non-const lvalue reference can't
        bind to a non-modifiable lvalue
    int& invalidRef2 { 0 }; // invalid: non-const lvalue reference can't
        bind to an rvalue
```

In most cases, a reference will only bind to an object whose type matches the referenced type. If you try to bind a reference to an object that does not match its referenced type, the compiler will try to implicitly convert the object to the referenced type and then bind the reference to that.

11.2.2 References can't be reseated

Once initialized, a reference cannot be reseated, meaning it cannot be changed to reference another object.

```
    int x { 5 };
    int y { 6 };

    int& ref { x }; // ref is now an alias for x

```

```cpp
    ref = y; // assigns 6 (the value of y) to x (the object being
        referenced by ref)
    // The above line does NOT change ref into a reference to variable y!

    std::cout << x << '\n'; // user is expecting this to print 5
```

Perhaps surprisingly, this prints: 6.

When a reference is evaluated in an expression, it resolves to the object it's referencing. So ref = y doesn't change ref to now reference y. Rather, because ref is an alias for x, the expression evaluates as if it was written x = y – and since y evaluates to value 6, x is assigned the value 6.

11.2.3 References and referents have independent lifetimes

With one exception, the lifetime of a reference and the lifetime of its referent are independent. In other words, both of the following are true:

- A reference can be destroyed before the object it is referencing.
- The object being referenced can be destroyed before the reference.

When a reference is destroyed before the referent, the referent is not impacted.

Dangling references: When an object being referenced is destroyed before a reference to it, the reference is left referencing an object that no longer exists. Such a reference is called a dangling reference. Accessing a dangling reference leads to undefined behavior.

Fortunately, dangling references are fairly easy to avoid

11.2.4 References aren't objects

Perhaps surprisingly, references are not objects in C++. Because references aren't objects, they can't be used anywhere an object is required. (e.g. you can't have a reference to a reference, since an lvalue reference must reference an identifiable object).
In cases where you need a reference that is an object or a reference that can be reseated, std::reference_wrapper provides a solution.

11.3 Lvalue reference to const

`https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/`

By using the const keyword when declaring an lvalue reference, we tell an lvalue reference to treat the object it is referencing as const. Such a reference is called an lvalue reference to a const value.

```cpp
    const int x { 5 };     // x is a non-modifiable lvalue
    const int& ref { x }; // okay: ref is a an lvalue reference to a const
        value
```

- Lvalue references to const can bind to non-modifiable lvalues.
- Lvalue references to const can also bind to modifiable lvalues. In such a case, the object being referenced is treated as const when accessed through the reference

```cpp
    int x { 5 };           // x is a modifiable lvalue
    const int& ref { x }; // okay: we can bind a const reference to a
        modifiable lvalue

    std::cout << ref << '\n'; // okay: we can access the object through
        our const reference
    ref = 7;                   // error: we can not modify an object
        through a const reference

    x = 6;                 // okay: x is a modifiable lvalue, we can still
        modify it through the original identifier
```

Perhaps surprisingly, lvalues references to const can also bind to rvalues:

```
    const int& ref { 5 }; // okay: 5 is an rvalue

    std::cout << ref << '\n'; // prints 5
```

Lvalue references to const can even bind to values of a different type, so long as those values can be implicitly converted to the reference type.

11.3.1 Const references bound to temporary objects extend the lifetime of the temporary object

Temporary objects are normally destroyed at the end of the expression in which they are created. However, if a temporary object created to hold rvalue was destroyed at the end of the expression that initializes a reference, the reference would be left dangling.
To avoid dangling references in such cases, C++ has a special rule: When a const lvalue reference is directly bound to a temporary object, the lifetime of the temporary object is extended to match the lifetime of the reference.

```
    const int& ref { 5 }; // The temporary object holding value 5 has its
        lifetime extended to match ref

    std::cout << ref << '\n'; // Therefore, we can safely use it here
```

11.3.2 Pass by lvalue reference

https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/

Most of the types provided by the standard library (such as std::string) are class types. Class types are usually expensive to copy. Whenever possible, we want to avoid making unnecessary copies of objects that are expensive to copy, especially when we will destroy those copies almost immediately.

One way to avoid making an expensive copy of an argument when calling a function is to use pass by reference instead of . When using pass by reference, we declare a function parameter as a reference type (or const reference type) rather than as a normal type.

```
    void printValue(std::string& y) // type changed to std::string&
    {
    std::cout << y << '\n';
    }Best practice

A pointer should either hold the address of a valid object, or be set to
    nullptr. That way we only need to test pointers for null, and can
    assume any non-null pointer is valid.Best practice

A pointer should either hold the address of a valid object, or be set to
    nullptr. That way we only need to test pointers for null, and can
    assume any non-null pointer is valid.
```

When an object is passed by value, the function parameter receives a copy of the argument. This means that any changes to the value of the parameter are made to the copy of the argument, not the argument itself.

However, since a reference acts identically to the object being referenced, when using pass by reference, any changes made to the reference parameter will affect the argument:

```
#include <iostream>

void addOne(int& y) // y is bound to the actual object x
{
    ++y; // this modifies the actual object x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x has been modified

    return 0;
}
```

11.4 Pass by const lvalue reference

https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/

If we make a reference parameter const, it will be able to bind to any type of argument:

```
    int x { 5 };
    printRef(x);   // ok: x is a modifiable lvalue, y binds to x

    const int z { 5 };
    printRef(z);   // ok: z is a non-modifiable lvalue, y binds to z

    printRef(5);   // ok: 5 is rvalue literal, y binds to temporary int
        object
```

Passing by const reference offers the same primary benefit as pass by non-const reference (avoiding making a copy of the argument), while also guaranteeing that the function can not change the value being referenced.

Now we can understand the motivation for allowing const lvalue references to bind to rvalues: without that capability, there would be no way to pass literals (or other rvalues) to functions that used pass by reference!

11.4.1 When to use pass by value vs pass by reference

For most C++ beginners, the choice of whether to use pass by value or pass by reference isn't very obvious. Fortunately, there's a straightforward rule of thumb that will serve you well in the majority cases.

- Fundamental types and enumerated types are cheap to copy, so they are typically passed by value.
- Class types can be expensive to copy (sometimes significantly so), so they are typically passed by const reference.

11.4.2 For function parameters, prefer `std::string_view` over `const std::string&` in most cases

In most cases, std::string_view is the better choice, as it can handle a wider range of argument types efficiently. A std::string_view parameter also allows the caller to pass in a substring without having to copy that substring into its own string first.

Chapter 12

Pointers

12.1 Introduction to pointers

https://www.learncpp.com/cpp-tutorial/introduction-to-pointers/

12.1.1 The address-of operator

Although the memory addresses used by variables aren't exposed to us by default, we do have access to this information. The address-of operator (&) returns the memory address of its operand. This is pretty straightforward:

```
    int x{ 5 };
    std::cout << x << '\n';  // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x

    std::cout << *(&x) << '\n'; // print the value at the memory address
        of variable x (parentheses not required, but make it easier to read)
```

12.1.2 Pointers

A **pointer** is an object that holds a memory address (typically of another variable) as its value. This allows us to store the address of some other object to use later.
Much like reference types are declared using an ampersand (&) character, pointer types are declared using an asterisk (*):

```
    int x { 5 };    // normal variable
    int& ref { x }; // a reference to an integer (bound to x)

    int* ptr;       // a pointer to an integer
```

On the author's machine, the above program printed:

```
5
0027FEA0
5
```

12.1.3 Pointer initialization

```
    int x{ 5 };

    int* ptr;         // an uninitialized pointer (holds a garbage address)
    int* ptr2{};      // a null pointer (we'll discuss these in the next
        lesson)
    int* ptr3{ &x }; // a pointer initialized with the address of variable
        x
```

Once we have a pointer holding the address of another object, we can then use the dereference operator (*) to access the value at that address. For example:

```
    int x{ 5 };
    std::cout << x << '\n'; // print the value of variable x

    int* ptr{ &x }; // ptr holds the address of x
    std::cout << *ptr << '\n'; // use dereference operator to print the
        value at the address that ptr is holding (which is x's address)
```

Much like the type of a reference has to match the type of object being referred to, the type of the pointer has to match the type of the object being pointed to:

```
    int* iPtr{ &i };      // ok: a pointer to an int can point to an int
        object
    int* iPtr2 { &d };    // not okay: a pointer to an int can't point to a
        double object
    double* dPtr{ &d };   // ok: a pointer to a double can point to a
        double object
    double* dPtr2{ &i }; // not okay: a pointer to a double can't point to
        an int object
```

With one exception that we'll discuss next lesson, initializing a pointer with a literal value is disallowed.

Best practice: Always initialize your pointers.

12.1.4 Pointers and assignment

We can use assignment with pointers in two different ways:

- To change what the pointer is pointing at (by assigning the pointer a new address)
- To change the value being pointed at (by assigning the dereferenced pointer a new value)

```
    int x{ 5 };
    int* ptr{ &x }; // ptr initialized to point at x

    std::cout << *ptr << '\n'; // print the value at the address being
        pointed to (x's address)

    int y{ 6 };
    ptr = &y; // // change ptr to point at y

    std::cout << *ptr << '\n'; // print the value at the address being
        pointed to (y's address)
```

Now let's look at how we can also use a pointer to change the value being pointed at:

```
    int x{ 5 };
    int* ptr{ &x }; // initialize ptr with address of variable x

    std::cout << x << '\n';    // print x's value
    std::cout << *ptr << '\n'; // print the value at the address that ptr
        is holding (x's address)

    *ptr = 6; // The object at the address held by ptr (x) assigned value 6
         (note that ptr is dereferenced here)

    std::cout << x << '\n';
    std::cout << *ptr << '\n'; // print the value at the address that ptr
        is holding (x's address)
```

Pointers behave much like lvalue references

```
    int x{ 5 };
    int& ref { x };  // get a reference to x
    int* ptr { &x }; // get a pointer to x

    std::cout << x;
    std::cout << ref;  // use the reference to print x's value (5)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (5)

    ref = 6; // use the reference to change the value of x
    std::cout << x;
    std::cout << ref;  // use the reference to print x's value (6)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (6)

    *ptr = 7; // use the pointer to change the value of x
    std::cout << x;
    std::cout << ref;  // use the reference to print x's value (7)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (7)
```

Pointers and references both provide a way to indirectly access another object. The primary difference is that with pointers, we need to explicitly get the address to point at, and we have to explicitly dereference the pointer to get the value. With references, the address-of and dereference happens implicitly.

There are some other differences between pointers and references worth mentioning:

- References must be initialized, pointers are not required to be initialized (but should be).
- References are not objects, pointers are.
- References can not be reseated (changed to reference something else), pointers can change what they are pointing at.
- References must always be bound to an object, pointers can point to nothing (we'll see an example of this in the next lesson).
- References are "safe" (outside of dangling references), pointers are inherently dangerous (we'll also discuss this in the next lesson).

The address-of operator returns a pointer

It's worth noting that the address-of operator (&) doesn't return the address of its operand as a literal. Instead, it returns a pointer containing the address of the operand, whose type is derived from the argument.

12.1.5 The size of pointers

The size of a pointer is dependent upon the architecture the executable is compiled for – a 32-bit executable uses 32-bit memory addresses – consequently, a pointer on a 32-bit machine is 32 bits (4 bytes). With a 64-bit executable, a pointer would be 64 bits (8 bytes). Note that this is true regardless of the size of the object being pointed to:

```
#include <iostream>

int main() // assume a 32-bit application
{
    char* chPtr{};        // chars are 1 byte
    int* iPtr{};          // ints are usually 4 bytes
    long double* ldPtr{}; // long doubles are usually 8 or 12 bytes

    std::cout << sizeof(chPtr) << '\n'; // prints 4
    std::cout << sizeof(iPtr) << '\n';  // prints 4
    std::cout << sizeof(ldPtr) << '\n'; // prints 4

    return 0;
}
```

The size of the pointer is always the same. This is because a pointer is just a memory address, and the number of bits needed to access a memory address is constant.

12.1.6 Dangling pointers

Much like a **dangling reference**, a dangling pointer is a pointer that is holding the address of an object that is no longer valid.

Dereferencing a dangling pointer will lead to undefined behavior, as you are trying to access an object that is no longer valid.
Here's an example of creating a dangling pointer:

```cpp
#include <iostream>

int main()
{
    int x{ 5 };
    int* ptr{ &x };

    std::cout << *ptr << '\n'; // valid

    {
        int y{ 6 };
        ptr = &y;

        std::cout << *ptr << '\n'; // valid
    } // y goes out of scope, and ptr is now dangling

    std::cout << *ptr << '\n'; // undefined behavior from dereferencing a
        dangling pointer

    return 0;
}
```

12.2 Null pointers

https://www.learncpp.com/cpp-tutorial/null-pointers/
Besides a memory address, there is one additional value that a pointer can hold: a null value. A null value (often shortened to null) is a special value that means something has no value. When a pointer is holding a null value, it means the pointer is not pointing at anything. Such a pointer is called a **null pointer**.

```cpp
    int* ptr {}; // ptr is now a null pointer, and is not holding an
        address
```

Much like the keywords true and false represent Boolean literal values, the nullptr keyword represents a null pointer literal. We can use nullptr to explicitly initialize or assign a pointer a null value.

```cpp
    int* ptr { nullptr }; // can use nullptr to initialize a pointer to be
        a null pointer
```

The nullptr keyword

Much like the keywords true and false represent Boolean literal values, the nullptr keyword represents a null pointer literal. We can use nullptr to explicitly initialize or assign a pointer a null value.

```cpp
int main()
{
    int* ptr { nullptr }; // can use nullptr to initialize a pointer to be
        a null pointer

    int value { 5 };
    int* ptr2 { &value }; // ptr2 is a valid pointer
    ptr2 = nullptr; // Can assign nullptr to make the pointer a null
        pointer

    someFunction(nullptr); // we can also pass nullptr to a function that
        has a pointer parameter

    return 0;
}
```

Dereferencing a null pointer results in undefined behaviour

Much like dereferencing a dangling (or wild) pointer leads to undefined behavior, dereferencing a null pointer also leads to undefined behaviour. In most cases, it will crash your application.

Checking for null pointers

Much like we can use a conditional to test Boolean values for true or false, we can use a conditional to test whether a pointer has value nullptr or not.
Similarly, pointers will also implicitly convert to Boolean values: a null pointer converts to Boolean value false, and a non-null pointer converts to Boolean value true. This allows us to skip explicitly testing for nullptr and just use the implicit conversion to Boolean to test whether a pointer is a null pointer. We can just do this to test whether a pointer has value nullptr or not:

```
    int x { 5 };
    int* ptr { &x };

    // pointers convert to Boolean false if they are null, and Boolean
        true if they are non-null
    if (ptr) // implicit conversion to Boolean
        std::cout << "ptr is non-null\n";
    else
        std::cout << "ptr is null\n";
```

Use nullptr to avoid dangling pointers

Above, we mentioned that dereferencing a pointer that is either null or dangling will result in undefined behavior. Therefore, we need to ensure our code does not do either of these things.
We can easily avoid dereferencing a null pointer by using a conditional to ensure a pointer is non-null before trying to dereference it.

```
// Assume ptr is some pointer that may or may not be a null pointer
if (ptr) // if ptr is not a null pointer
    std::cout << *ptr << '\n'; // okay to dereference
else
    // do something else that doesn't involve dereferencing ptr (print an
        error message, do nothing at all, etc...)
```

When an object is destroyed, any pointers to that object will be left dangling. Such pointers are not nulled automatically! It is the programmer's responsibility to ensure that all pointers to an object that has just been destroyed are properly set to nullptr.

Best practice : A pointer should either hold the address of a valid object, or be set to nullptr. That way we only need to test pointers for null, and can assume any non-null pointer is valid.

Best practice : Favor references over pointers unless the additional capabilities provided by pointers are needed.

Pointers and references both give us the ability to access some other object indirectly.
Pointers have the additional abilities of being able to change what they are pointing at, and to be pointed at null. However, these pointer abilities are also inherently dangerous: A null pointer runs the risk of being dereferenced, and the ability to change what a pointer is pointing at can make creating dangling pointers easier:

```
int main()
{
    int* ptr { };

    {
        int x{ 5 };
        ptr = &x; // assign the pointer to an object that will be
            destroyed (not possible with a reference)
    } // ptr is now dangling and pointing to invalid object

    if (ptr) // condition evaluates to true because ptr is not nullptr
        std::cout << *ptr; // undefined behavior

    return 0;
}
```

Since references can't be bound to null, we don't have to worry about null references. And because references must be bound to a valid object upon creation and then can not be reseated, dangling references are harder to create.
Because they are safer, references should be favored over pointers, unless the additional capabilities provided by pointers are required.

A joke about null-pointers

Did you hear the joke about the null pointer?
That's okay, you wouldn't get dereference.

12.2.1 Pointers and const

https://www.learncpp.com/cpp-tutorial/pointers-and-const/

Pointer to const value

A pointer to a const value is a (non-const) pointer that points to a constant value. To declare a pointer to a const value, use the const keyword before the pointer's data type:

```
    const int x{ 5 };
    const int* ptr { &x }; // okay: ptr is pointing to a "const int"

    *ptr = 6; // not allowed: we can't change a const value
```

However, because a pointer to const is not const itself (it just points to a const value), we can change what the pointer is pointing at by assigning the pointer a new address:

```
    const int x{ 5 };
    const int* ptr { &x }; // ptr points to const int x

    const int y{ 6 };
    ptr = &y; // okay: ptr now points at const int y
```

Just like a reference to const, a pointer to const can point to non-const variables too. A pointer to const treats the value being pointed to as constant, regardless of whether the object at that address was initially defined as const or not:

```
    int x{ 5 }; // non-const
    const int* ptr { &x }; // ptr points to a "const int"

    *ptr = 6;  // not allowed: ptr points to a "const int" so we can't
        change the value through ptr
    x = 6; // allowed: the value is still non-const when accessed through
        non-const identifier x
```

Const pointers

To declare a const pointer, use the const keyword after the asterisk in the pointer declaration:

```
    int x{ 5 };
    int* const ptr { &x }; // const after the asterisk means this is a
        const pointer
```

Just like a normal const variable, a const pointer must be initialized upon definition, and this value can't be changed via assignment.

```cpp
    int x{ 5 };
    int y{ 6 };

    int* const ptr { &x }; // okay: the const pointer is initialized to
        the address of x
    ptr = &y; // error: once initialized, a const pointer can not be
        changed.
```

However, because the value being pointed to is non-const, it is possible to change the value being pointed to via dereferencing the const pointer:

```cpp
    int x{ 5 };
    int* const ptr { &x }; // ptr will always point to x

    *ptr = 6; // okay: the value being pointed to is non-const
```

Const pointer to a const value

Finally, it is possible to declare a const pointer to a const value by using the const keyword both before the type and after the asterisk.

```cpp
    const int* const ptr { &value }; // a const pointer to a const value
```

12.2.2 Pass by address

https://www.learncpp.com/cpp-tutorial/pass-by-address/

Pass by address

With pass by address, instead of providing an object as an argument, the caller provides an object's address (via a pointer). This pointer (holding the address of the object) is copied into a pointer parameter of the called function. The function can then dereference that pointer to access the object whose address was passed.

```cpp
void printByValue(std::string val) // The function parameter is a copy of
    str
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const std::string& ref) // The function parameter is
    a reference that binds to str
{
    std::cout << ref << '\n'; // print the value via the reference
}

void printByAddress(const std::string* ptr) // The function parameter is a
    pointer that holds the address of str
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced
        pointer
}
```

Let's explore the pass by address version in more detail.

First, because we want our printByAddress() function to use pass by address, we've made our function parameter a pointer named ptr. Since printByAddress() will use ptr in a read-only manner, ptr is a pointer to a const value.

```cpp
void printByAddress(const std::string* ptr)
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced
        pointer
}
```

Second, when the function is called, we can't just pass in the str object – we need to pass in the address of str. The easiest way to do that is to use the address-of operator (&) to get a pointer holding the address of str:

Although we use the address-of operator in the above example to get the address of str, if we already had a pointer variable holding the address of str, we could use that instead:

```cpp
printByAddress(&str); // use address-of operator (&) to get pointer
    holding address of str
printByAddress(ptr); // pass str by address, does not make a copy of str
```

Just like pass by reference, pass by address is fast, and avoids making a copy of the argument object.

Pass by address allows the function to modify the argument's value

When we pass an object by address, the function receives the address of the passed object, which it can access via dereferencing. If the function parameter is a pointer to non-const, the function can modify the argument via the pointer parameter:

```cpp
void changeValue(int* ptr) // note: ptr is a pointer to non-const in this
    example
{
    *ptr = 6; // change the value to 6
}
```

Best practice : Prefer pointer-to-const function parameters over pointer-to-non-const function parameters, unless the function needs to modify the object passed in. Do not make function parameters const pointers unless there is some specific reason to do so.

Prefer pass by (const) reference : Pass by reference has the same benefits as pass by address without the risk of inadvertently dereferencing a null pointer.

Pass by const reference has a few other advantages over pass by address.
First, because an object being passed by address must have an address, only lvalues can be passed by address (as rvalues don't have addresses). Pass by const reference is more flexible, as it can accept lvalues and rvalues: Second, the syntax for pass by reference is natural, as we can just pass in literals or objects. With pass by address, our code ends up littered with ampersands (&) and asterisks (*).
In modern C++, most things that can be done with pass by address are better accomplished through other methods. Follow this common maxim: "Pass by reference when you can, pass by address when you must".

12.2.3 Pass by address for "optional" arguments

https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/
One of the more common uses for pass by address is to allow a function to accept an "optional" argument. This is easier to illustrate by example than to describe:

```cpp
#include <iostream>

void printIDNumber(const int *id=nullptr)
{
    if (id)
        std::cout << "Your ID number is " << *id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(&userid); // we know the user's ID now

    return 0;
}
```

However, in many cases, function overloading is a better alternative to achieve the same result:

```
#include <iostream>

void printIDNumber()
{
    std::cout << "Your ID is not known\n";
}

void printIDNumber(int id)
{
    std::cout << "Your ID is " << id << "\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(userid); // we know the user is 34

    printIDNumber(62); // now also works with rvalue arguments

    return 0;
}
```

This has a number of advantages: we no longer have to worry about null dereferences, and we can pass in literals or other rvalues as an argument.

Changing what a pointer parameter points at

When we pass an address to a function, that address is copied from the argument into the pointer parameter (which is fine, because copying an address is fast). Now consider the following program:

```
#include <iostream>

// [[maybe_unused]] gets rid of compiler warnings about ptr2 being set but
    not used
void nullify([[maybe_unused]] int* ptr2)
{
    ptr2 = nullptr; // Make the function parameter a null pointer
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr points to x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");
    return 0;
}
```

This program prints:

```
ptr is non-null
ptr is non-null
```

ptr is non-null As you can see, changing the address held by the pointer parameter had no impact on the address held by the argument (ptr still points at x). When function nullify() is called, ptr2 receives a copy of the address passed in (in this case, the address held by ptr, which is the address of x). When the function changes what ptr2 points at, this only affects the copy held by ptr2.

12.2.4 Pass by address by reference?

Yup, it's a thing. Just like we can pass a normal variable by reference, we can also pass pointers by reference. Here's the same program as above with ptr2 changed to be a reference to an address:

```cpp
#include <iostream>

void nullify(int*& refptr) // refptr is now a reference to a pointer
{
    refptr = nullptr; // Make the function parameter a null pointer
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr points to x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");
    return 0;
}
```

This program prints:

```
ptr is non-null
ptr is null
```

Because refptr is now a reference to a pointer, when ptr is passed as an argument, refptr is bound to ptr. This means any changes to refptr are made to ptr.
Note: Because references to pointers are fairly uncommon, it can be easy to mix up the syntax (is it int*& or int&*?). The good news is that if you do it backwards, the compiler will error because you can't have a pointer to a reference (because pointers must hold the address of an object, and references aren't objects). Then you can switch it around.

There is only pass by value

Now that you understand the basic differences between passing by reference, address, and value, let's get reductionist for a moment. :)
While the compiler can often optimize references away entirely, there are cases where this is not possible and a reference is actually needed. References are normally implemented by the compiler using pointers. This means that behind the scenes, pass by reference is essentially just a pass by address.
And in the previous lesson, we mentioned that pass by address just copies an address from the caller to the called function – which is just passing an address by value.
Therefore, we can conclude that C++ really passes everything by value! The properties of pass by address (and reference) come solely from the fact that we can dereference the passed address to change the argument, which we can not do with a normal value parameter!

12.3 Return by reference and return by address

https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/

12.3.1 Return by reference

In cases where we're passing a class type back to the caller, we may (or may not) want to return by reference instead. **Return by reference** returns a reference that is bound to the object being returned, which avoids making a copy of the return value. To return by reference, we simply define the return value of the function to be a reference type:

```cpp
std::string&       returnByReference(); // returns a reference to an
    existing std::string (cheap)
const std::string& returnByReferenceToConst(); // returns a const
    reference to an existing std::string (cheap)
```

The object being returned by reference must exist after the function returns

Warning: Objects returned by reference must live beyond the scope of the function returning the reference, or a dangling reference will result. Never return a (non-static) local variable or temporary by reference.

Using return by reference has one major caveat: the programmer must be sure that the object being referenced outlives the function returning the reference. Otherwise, the reference being returned will be left dangling (referencing an object that has been destroyed), and use of that reference will result in undefined behavior. Consider this example:

```cpp
#include <iostream>
#include <string>

const std::string& getProgramName()
{
    const std::string programName { "Calculator" }; // now a non-static
        local variable, destroyed when function ends

    return programName;
}

int main()
{
    std::cout << "This program is named " << getProgramName(); //
        undefined behavior

    return 0;
}
```

Lifetime extension doesn't work across function boundaries

Let's take a look at an example where we return a temporary by reference:

```cpp
#include <iostream>

const int& returnByConstReference(const int& ref)
{
    return ref;
}

int main()
{
    // case 1: direct binding
    const int& ref1 { 5 }; // extends lifetime
    std::cout << ref1 << '\n'; // okay

    // case 2: indirect binding
    const int& ref2 { returnByConstReference(5) }; // binds to dangling
        reference
    std::cout << ref2 << '\n'; // undefined behavior

    return 0;
}
```

In case 2, a temporary object is created to hold value 5, which function parameter ref binds to. The function just returns this reference back to the caller, which then uses the reference to initialize ref2. Because this is not a direct binding to the temporary object (as the refrence was bounced through a function), lifetime extension doesn't apply. This leaves ref2 dangling, and its subsequent use is undefined behavior.

Best practice: Avoid returning references to non-const local static variables.

It's okay to return reference parameters by reference

There are quite a few cases where returning objects by reference makes sense, and we'll encounter many of those in future lessons. However, there is one useful example that we can show now.
If a parameter is passed into a function by reference, it's safe to return that parameter by reference. This makes sense: in order to pass an argument to a function, the argument must exist in the scope of the caller. When the called function returns, that object must still exist in the scope of the caller.
Here is a simple example of such a function:

```cpp
#include <iostream>
#include <string>

// Takes two std::string objects, returns the one that comes first
    alphabetically
const std::string& firstAlphabetical(const std::string& a, const
    std::string& b)
{
  return (a < b) ? a : b; // We can use operator< on std::string to
      determine which comes first alphabetically
}

int main()
{
  std::string hello { "Hello" };
  std::string world { "World" };

  std::cout << firstAlphabetical(hello, world) << '\n';

  return 0;
}
```

It's okay for an rvalue passed by const reference to be returned by const reference

12.3.2 Return by address

Return by address works almost identically to return by reference, except a pointer to an object is returned instead of a reference to an object. Return by address has the same primary caveat as return by reference – the object being returned by address must outlive the scope of the function returning the address, otherwise the caller will receive a dangling pointer.
The major advantage of return by address over return by reference is that we can have the function return nullptr if there is no valid object to return. For example, let's say we have a list of students that we want to search. If we find the student we are looking for in the list, we can return a pointer to the object representing the matching student. If we don't find any students matching, we can return nullptr to indicate a matching student object was not found.
The major disadvantage of return by address is that the caller has to remember to do a nullptr check before dereferencing the return value, otherwise a null pointer dereference may occur and undefined behavior will result. Because of this danger, return by reference should be preferred over return by address unless the ability to return "no object" is needed.

Best Practice: Prefer return by reference over return by address unless the ability to return "no object" (using nullptr) is important.

12.3.3 Type deduction with pointers, references, and const

We already noted that by default, type deduction will drop const from types.
Const (or constexpr) can be reapplied by adding the const (or constexpr) qualifier to the definition of the deduced type.

Type deduction drops references

In addition to dropping const, type deduction will also drop references:

```
#include <string>

std::string& getRef(); // some function that returns a reference

int main()
{
    auto ref { getRef() }; // type deduced as std::string (not
        std::string&)

    return 0;
}
```

In the above example, variable ref is using type deduction. Although function getRef() returns a std::string&, the reference qualifier is dropped, so the type of ref is deduced as std::string.

```
#include <string>

std::string& getRef(); // some function that returns a reference

int main()
{
    auto ref1 { getRef() };  // std::string (reference dropped)
    auto& ref2 { getRef() }; // std::string& (reference dropped, reference
        reapplied)

    return 0;
}
```

Top-level const and low-level const

A **top-level const** is a const qualifier that applies to an object itself. For example:

```
const int x;    // this const applies to x, so it is top-level
int* const ptr; // this const applies to ptr, so it is top-level
// references don't have a top-level const syntax, as they are implicitly
    top-level const
```

In contrast, a **low-level const** is a const qualifier that applies to the object being referenced or pointed to:

```
const int& ref; // this const applies to the object being referenced, so
    it is low-level
const int* ptr; // this const applies to the object being pointed to, so
    it is low-level
```

A reference to a const value is always a low-level const. A pointer can have a top-level, low-level, or both kinds of const:

```
const int* const ptr; // the left const is low-level, the right const is
    top-level
```

Note: When we say that type deduction drops const qualifiers, it only drops top-level consts. Low-level consts are not dropped.

Type deduction and pointers

Unlike references, type deduction does not drop pointers:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    auto ptr1{ getPtr() }; // std::string*

    return 0;
}
```

We can also use an asterisk in conjunction with pointer type deduction (auto*) to make it clearer that the deduced type is a pointer:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    auto ptr1{ getPtr() };  // std::string*
    auto* ptr2{ getPtr() }; // std::string*

    return 0;
}
```

And it is generally better to do so if we want auto to resolve as a pointer for reason we are not going to dive into.

Chapter 13

Compound types

13.1 Introduction to program-defined types

`https://www.learncpp.com/cpp-tutorial/introduction-to-program-defined-user-defined-types/`
We already introduced the challenge of wanting to store a fraction, which has a numerator and denominator that are conceptually linked together. In that lesson, we discussed some of the challenges with using two separate integers to store a fraction's numerator and denominator independently.
C++ solves such problems allowing the creation of entirely new, custom types that we can use in our programs! Such types are called **user-defined types**. However, as we will discuss later in this lesson, we'll prefer the term **program-defined types** for any such types that we create for use in our own programs.
C++ has two different categories of compound types that can be used to create program-defined types:

- Enumerated types (including unscoped and scoped enumerations)
- Class types (including structs, classes, and unions).

13.1.1 Defining program-defined types

Just like type aliases, program-defined types must also be defined and given a name before they can be used. The definition for a program-defined type is called a **type definition**.
Although we haven't covered what a struct is yet, here's an example showing the definition of custom Fraction type and an instantiation of an object using that type:

```cpp
// Define a program-defined type named Fraction so the compiler understands what a Fraction is
// (we'll explain what a struct is and how to use them later in this chapter)
// This only defines what a Fraction type looks like, it doesn't create one
struct Fraction
{
  int numerator {};
  int denominator {};
};

// Now we can make use of our Fraction type
int main()
{
  Fraction f { 3, 4 }; // this actually instantiates a Fraction object named f

  return 0;
}
```

In this example, we're using the struct keyword to define a new program-defined type named Fraction (in the global scope, so it can be used anywhere in the rest of the file). This doesn't allocate any memory – it just tells the compiler what a Fraction looks like, so we can allocate objects of a Fraction type later. Then, inside main(), we instantiate (and initialize) a variable of type Fraction named f.
Program-defined type definitions must end in a semicolon. Failure to include the semicolon at the end of a type definition is a common programmer error, and one that can be hard to debug because the compiler may error on the line after the type definition.

13.1.2 Naming program-defined types

By convention, program-defined types are named starting with a capital letter and don't use a suffix

```
Fraction fraction {}; // Instantiates a variable named fraction of type
    Fraction
```

13.1.3 Using program-defined types throughout a multi-file program

Every code file that uses a program-defined type needs to see the full type definition before it is used. A forward declaration is not sufficient. This is required so that the compiler knows how much memory to allocate for objects of that type.
To propagate type definitions into the code files that need them, program-defined types are typically defined in header files, and then `#included` into any code file that requires that type definition. These header files are typically given the same name as the program-defined type (e.g. a program-defined type named Fraction would be defined in Fraction.h)
Fraction.h:

```
#ifndef FRACTION_H
#define FRACTION_H

// Define a new type named Fraction
// This only defines what a Fraction looks like, it doesn't create one
// Note that this is a full definition, not a forward declaration
struct Fraction
{
  int numerator {};
  int denominator {};
};

#endif
```

Franction.cpp will include only the definition of member functions or other stuff that we will introduce later.

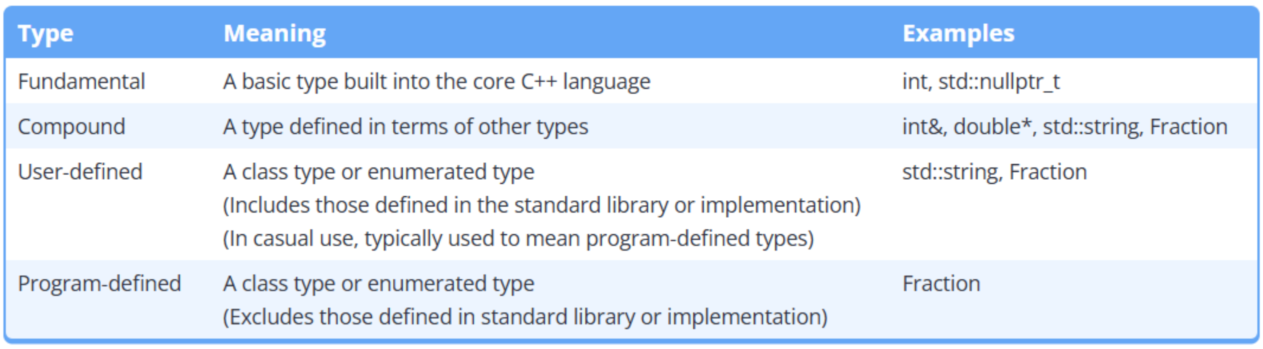
Type definitions are partially exempt from the one-definition rule (ODR)

For user-defined types using forward declarations doesn't work, because the compiler typically needs to see the full definition to use a given type. We must be able to propagate the full type definition to each code file that needs it.
For this reason, types are partially exempt from the one-definition rule: a given type is allowed to be defined in multiple code files.
There are two caveats that are worth knowing about.

- First, you can still only have one type definition per code file (this usually isn't a problem since header guards will prevent this).
- Second, all of the type definitions for a given type must be identical, otherwise undefined behavior will result.

Nomenclature: user-defined types vs program-defined types

C++20 language standard helpfully defines the term "program-defined type" to mean class types and enumerated types that are not defined as part of the standard library, implementation, or core language. In other words, "program-defined types" only include class types and enum types that are defined by us (or a third-party library).



Type	Meaning	Examples
Fundamental	A basic type built into the core C++ language	int, std::nullptr_t
Compound	A type defined in terms of other types	int&, double*, std::string, Fraction
User-defined	A class type or enumerated type (Includes those defined in the standard library or implementation) (In casual use, typically used to mean program-defined types)	std::string, Fraction
Program-defined	A class type or enumerated type (Excludes those defined in standard library or implementation)	Fraction

13.2 Structs

`https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/`
There are many instances in programming where we need more than one variable in order to represent something of interest.
Lets say we want to write a program where we need to store information about the employees in a company. We might be interested in keeping track of attributes such as the employee's name, title, age, employee id, manager id, wage, birthday, hire date, etc...
If we were to use independent variables to track all of this information, that might look something like this:

```
std::string name;
std::string title;
int age;
int id;
int managerId;
double wage;
int birthdayYear;
int birthdayMonth;
int birthdayDay;
int hireYear;
int hireMonth;
int hireDay;
```

However, there are a number of problems with this approach. First, it's not immediately clear whether these variables are actually related or not (you'd have to read comments, or see how they are used in context). Second, there are now 12 variables to manage. If we wanted to pass this employee to a function, we'd have to pass 12 arguments (and in the correct order), which would make a mess of our function prototypes and function calls. And since a function can only return a single value, how would a function even return an employee?
Fortunately, C++ comes with two compound types designed to solve such challenges: structs (short for structure). These make management of related sets of variables much simpler!

A **struct** is a program-defined data type that allows us to bundle multiple variables together into a single type.

Note: A struct is a class type (as are classes and unions). As such, anything that applies to class types applies to structs.

13.2.1 Definition

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};
```

The struct keyword is used to tell the compiler that we're defining a struct, which we've named Employee (since program-defined types are typically given names starting with a capital letter).

Then, inside a pair of curly braces, we define the variables that each Employee object will contain. In this example, each Employee we create will have 3 variables: an int id, an int age, and a double wage. The variables that are part of the struct are called **data members** (or **member variables**).
Just like we use an empty set of curly braces to value initialize normal variables, the empty curly braces after each member variable ensures that the member variables inside our Employee are value initialized when an Employee is created.
Finally, we end the type definition with a semicolon.
As a reminder, Employee is just a type definition – no objects are actually created at this point.

13.2.2 Defining struct objects

In order to use the Employee type, we simply define a variable of type Employee:

```cpp
Employee joe {}; // Employee is the type, joe is the variable name
```

This defines a variable of type Employee named joe. When the code is executed, an Employee object is instantiated that contains the 3 data members. The empty braces ensures our object is value-initialized.
Just like any other type, it is possible to define multiple variables of the same struct type:

```cpp
Employee joe {}; // create an Employee struct for Joe
Employee frank {}; // create an Employee struct for Frank
```

13.2.3 Accessing members

In the above example, the name joe refers to the entire struct object (which contains the member variables). To access a specific member variable, we use the member selection operator (operator.) in between the struct variable name and the member name. For example, to access Joe's age member, we'd use joe.age.
Struct member variables work just like normal variables, so it is possible to do normal operations on them, including assignment, arithmetic, comparison, etc…

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe {};

    joe.age = 32;  // use member selection operator (.) to select the age
        member of variable joe

    std::cout << joe.age << '\n'; // print joe's age

    return 0;
}
```

One of the biggest advantages of structs is that we only need to create one new name per struct variable (the member names are fixed as part of the struct type definition). In the following example, we instantiate two Employee objects: joe and frank.

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
```

```cpp
{
    Employee joe {};
    joe.id = 14;
    joe.age = 32;
    joe.wage = 60000.0;

    Employee frank {};
    frank.id = 15;
    frank.age = 28;
    frank.wage = 45000.0;

    int totalAge { joe.age + frank.age };
    std::cout << "Joe and Frank have lived " << totalAge << " total
       years\n";

    if (joe.wage > frank.wage)
        std::cout << "Joe makes more than Frank\n";
    else if (joe.wage < frank.wage)
        std::cout << "Joe makes less than Frank\n";
    else
        std::cout << "Joe and Frank make the same amount\n";

    // Frank got a promotion
    frank.wage += 5000.0;

    // Today is Joe's birthday
    ++joe.age; // use pre-increment to increment Joe's age by 1

    return 0;
}
```

13.2.4 Aggregate

In general programming, an **aggregate data type** (also called an **aggregate**) is any type that can contain multiple data members. Some types of aggregates allow members to have different types (e.g. structs), while others require that all members must be of a single type (e.g. arrays).

Aggregate initialization of a struct

Aggregates use a form of initialization called **aggregate initialization**, which allows us to directly initialize the members of aggregates. To do this, we provide an **initializer list** as an initializer, which is just a braced list of comma-separated values.
There are 2 primary forms of aggregate initialization:

```cpp
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee frank = { 1, 32, 60000.0 }; // copy-list initialization using
       braced list
    Employee joe { 2, 28, 45000.0 };     // list initialization using
       braced list (preferred)

    return 0;
}
```

Best practice: Prefer the (non-copy) braced list form when initializing aggregates.

Overloading operator<< to print a struct

It is possible to overload operator<< for structs (we will explore this concept further in the future).

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

std::ostream& operator<<(std::ostream& out, const Employee& e)
{
    out << "id: " << e.id << " age: " << e.age << " wage: " << e.wage;
    return out;
}

int main()
{
    Employee joe { 2, 28 }; // joe.wage will be value-initialized to 0.0
    std::cout << joe << '\n';

    return 0;
}
```

This prints:

```
id: 2 age: 28 wage: 0
```

13.2.5 Const structs

Variables of a struct type can be const (or constexpr), and just like all const variables, they must be initialized.

```cpp
struct Rectangle
{
    double length {};
    double width {};
};

int main()
{
    const Rectangle unit { 1.0, 1.0 };
    const Rectangle zero { }; // value-initialize all members

    return 0;
}
```

13.2.6 Designated initializers

When initializing a struct from a list of values, the initializers are applied to the members in order of declaration.

```cpp
struct Foo
{
    int a {};
    int c {};
};

int main()
{
    Foo f { 1, 3 }; // f.a = 1, f.c = 3

    return 0;
}
```

Now consider what would happen if you were to update this struct definition to add a new member that is not the last member:

```cpp
struct Foo
{
    int a {};
    int b {}; // just added
    int c {};
};

int main()
{
    Foo f { 1, 3 }; // now, f.a = 1, f.b = 3, f.c = 0

    return 0;
}
```

Now all your initialization values have shifted, and worse, the compiler may not detect this as an error (after all, the syntax is still valid).
To help avoid this, C++20 adds a new way to initialize struct members called designated initializers. Designated initializers allow you to explicitly define which initialization values map to which members. The members can be initialized using list or copy initialization, and must be initialized in the same order in which they are declared in the struct, otherwise a warning or error will result. Members not designated an initializer will be value initialized.

```cpp
struct Foo
{
    int a{ };
    int b{ };
    int c{ };
};

int main()
{
    Foo f1{ .a{ 1 }, .c{ 3 } }; // ok: f1.a = 1, f1.b = 0 (value
        initialized), f1.c = 3
    Foo f2{ .a = 1, .c = 3 };   // ok: f2.a = 1, f2.b = 0 (value
        initialized), f2.c = 3
    Foo f3{ .b{ 2 }, .a{ 1 } }; // error: initialization order does not
        match order of declaration in struct

    return 0;
}
```

Designated initializers are nice because they provide some level of self-documentation and help ensure you don't inadvertently mix up the order of your initialization values. However, designated initializers also clutter up the initializer list significantly, so we won't recommend their use as a best practice at this time.
Also, because there's no enforcement that designated initializers are being used consistently everywhere an aggregate is initialized, it's a good idea to avoid adding new members to the middle of an existing aggregate definition, to avoid the risk of initializer shifting.

13.2.7 Assignment with an initializer list

We can assign values to members of structs individually:

```cpp
    joe.age  = 33;      // Joe had a birthday
    joe.wage = 66000.0; // and got a raise
```

This is fine for single members, but not great when we want to update many members. Similar to initializing a struct with an initializer list, you can also assign values to structs using an initializer list (which does memberwise assignment):

```cpp
    Employee joe { 1, 32, 60000.0 };
    joe = { joe.id, 33, 66000.0 }; // Joe had a birthday and got a raise
```

Initializing a struct with another struct of the same type

A struct may also be initialized using another struct of the same type:

```cpp
struct Foo
{
    int a{};
    int b{};
    int c{};
};

int main()
{
    Foo foo { 1, 2, 3 };

    Foo x = foo; // copy-initialization
    Foo y(foo);  // direct-initialization
    Foo z {foo}; // direct-list-initialization

    std::cout << x << '\n';
    std::cout << y << '\n';
    std::cout << z << '\n';

    return 0;
}
```

13.2.8 Default member initialization

When we define a struct (or class) type, we can provide a default initialization value for each member as part of the type definition. For members not marked as static, this process is sometimes called non-static member initialization. The initialization value is called a default member initializer.

```cpp
struct Something
{
    int x;       // no initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s1; // s1.x is uninitialized, s1.y is 0, and s1.z is 2

    return 0;
}
```

Explicit initialization values take precedence over default values

```cpp
struct Something
{
    int x;       // no default initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s2 { 5, 6, 7 }; // use explicit initializers for s2.x, s2.y,
        and s2.z (no default values are used)

    return 0;
}
```

If the number of initialization values is fewer than the number of members, then all remaining members will be value-initialized. However, if a default member initializer is provided for a given member, that default member initializer will be used instead.

```cpp
struct Something
{
    int x;        // no default initialization value (bad)
    int y {};     // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s3 {}; // value initialize s3.x, use default values for s3.y
        and s3.z

    return 0;
}
```

Best practice: Provide a default value for all members. This ensures that your members will be initialized even if the variable definition doesn't include an initializer list.

Default initialization vs value initialization for aggregates

Preferring value initialization has one more benefit – it's consistent with how we initialize objects of other types. Consistency helps prevent errors.

Best practice: For aggregates, prefer value initialization (with an empty braces initializer) to default initialization (with no braces).

13.3 Passing and returning structs

A big advantage of using structs over individual variables is that we can pass the entire struct to a function that needs to work with the members. Structs are generally passed by reference (typically by const reference) to avoid making copies.

```cpp
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

void printEmployee(const Employee& employee) // note pass by reference here
{
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}
```

Passing temporary structs

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

void printEmployee(const Employee& employee) // note pass by reference here
{
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}

int main()
{
    // Print Joe's information
    printEmployee(Employee { 14, 32, 24.15 }); // construct a temporary
        Employee to pass to function (type explicitly specified) (preferred)

    std::cout << '\n';

    // Print Frank's information
    printEmployee({ 15, 28, 18.27 }); // construct a temporary Employee to
        pass to function (type deduced from parameter)

    return 0;
}
```

We can create a temporary Employee in two ways. In the first call, we use the syntax Employee 14, 32, 24.15 . This tells the compiler to create an Employee object and initialize it with the provided initializers. This is the preferred syntax because it makes clear what kind of temporary object we are creating, and there is no way for the compiler to misinterpret our intentions.
In the second call, we use the syntax 15, 28, 18.27 . The compiler is smart enough to understand that the provided arguments must be converted to an Employee so that the function call will succeed. Note that this form is considered an implicit conversion, so it will not work in cases where only explicit conversions are acceptable.

13.3.1 Returning structs

Consider the case where we have a function that needs to return a point in 3-dimensional Cartesian space. Such a point has 3 attributes: an x-coordinate, a y-coordinate, and a z-coordinate. But functions can only return one value. So how do we return all 3 coordinates back the user?
One common way is to return a struct:

```
struct Point3d
{
    double x { 0.0 };
    double y { 0.0 };
    double z { 0.0 };
};

Point3d getZeroPoint()
{
    // We can create a variable and return the variable (we'll improve
        this below)
    Point3d temp { 0.0, 0.0, 0.0 };
    return temp;
}
```

In the getZeroPoint() function above, we create a new named object (temp) just so we could return it, the name of the object (temp) doesn't really provide any documentation value here.
We can make our function slightly better by returning a temporary (unnamed/anonymous) object instead:

```
Point3d getZeroPoint()
{
    return Point3d { 0.0, 0.0, 0.0 }; // return an unnamed Point3d
}
```

In this case, a temporary Point3d is constructed, copied back to the caller, and then destroyed at the end of the expression. Note how much cleaner this is (one line vs two, and no need to understand whether temp is used more than once).
In the case where the function has an explicit return type (e.g. Point3d), we can even omit the type in the return statement:

```
Point3d getZeroPoint()
```

```
{
    // We already specified the type at the function declaration
    // so we don't need to do so here again
    return { 0.0, 0.0, 0.0 }; // return an unnamed Point3d
}
```

Also note that since in this case we're returning all zero values, we can use empty braces to return a value-initialized Point3d:

```
Point3d getZeroPoint()
{
    // We can use empty curly braces to value-initialize all members
    return {};
}
```

13.4 Member selection for pointers and references

Since references to an object act just like the object itself, we can also use the member selection operator (.) to select a member from a reference to a struct:

13.4.1 Member selection for pointers to structs

However, the member selection operator (.) can't be used directly on a pointer to a struct.
With normal variables or references, we can access objects directly. However, because pointers hold addresses, we first need to dereference the pointer to get the object before we can do anything with it. So one way to access a member from a pointer to a struct is as follows:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << (*ptr).id << '\n'; // Not great but works: First
        dereference ptr, then use member selection

    return 0;
}
```

However, this is a bit ugly, especially because we need to parenthesize the dereference operation so it will take precedence over the member selection operation.
To make for a cleaner syntax, C++ offers a **member selection from pointer operator** (-¿) (also sometimes called the arrow operator) that can be used to select members from a pointer to an object:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
```

```
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << ptr->id << '\n'; // Better: use -> to select member from
        pointer to object

    return 0;
}
```

This member selection from pointer operator (-¿) works identically to the member selection operator (.) but does an implicit dereference of the pointer object before selecting the member. Thus ptr-¿id is equivalent to (*ptr).id.
This arrow operator is not only easier to type, but is also much less prone to error because the indirection is implicitly done for you, so there are no precedence issues to worry about. Consequently, when doing member access through a pointer, always use the -¿ operator instead of the . operator.

A **Best practice:** when using a pointer to access a member, use the member selection from pointer operator (->).

If the member accessed via operator-¿ is a pointer to a class type, operator-¿ can be applied again in the same expression to access the member of that class type.

```
#include <iostream>

struct Point
{
    double x {};
    double y {};
};

struct Triangle
{
    Point* a {};
    Point* b {};
    Point* c {};
};

int main()
{
    Point a {1,2};
    Point b {3,7};
    Point c {10,2};

    Triangle tr { &a, &b, &c };
    Triangle* ptr {&tr};

    // ptr is a pointer to a Triangle, which contains members that are
        pointers to a Point
    // To access member y of Point c of the Triangle pointed to by ptr,
        the following are equivalent:

    // access via operator.
    std::cout << (*(*ptr).c).y << '\n'; // ugly!

    // access via operator->
    std::cout << ptr -> c -> y << '\n'; // much nicer
}
```

The member selection operator is always applied to the currently selected variable. If you have a mix of pointers and normal member variables, you can see member selections where . and -¿ are both used in sequence:

```cpp
#include <iostream>
#include <string>

struct Paw
{
    int claws{};
};

struct Animal
{
    std::string name{};
    Paw paw{};
};

int main()
{
    Animal puma{ "Puma", { 5 } };

    Animal* ptr{ &puma };

    // ptr is a pointer, use ->
    // paw is not a pointer, use .

    std::cout << (ptr->paw).claws << '\n';

    return 0;
}
```

13.5 Class templates

For example, let's say we're writing a program where we need to work with pairs of int values, and need to determine which of the two numbers is larger. We might write a program like this:

```cpp
struct Pair
{
    int first{};
    int second{};
};

constexpr int max(Pair p) // pass by value because Pair is small
{
    return (p.first < p.second ? p.second : p.first);
}
```

Later, we discover that we also need pairs of double values. So we update our program to the following:

```cpp
#include <iostream>

struct Pair
{
    int first{};
    int second{};
};

struct Pair // compile error: erroneous redefinition of Pair
{
    double first{};
    double second{};
};

constexpr int max(Pair p)
{
    return (p.first < p.second ? p.second : p.first);
}

```

```
constexpr double max(Pair p) // compile error: overloaded function differs
    only by return type
{
    return (p.first < p.second ? p.second : p.first);
}
```

Fortunately, we can do better.
Much like a function template is a template definition for instantiating functions, a **class template** is a template definition for instantiating class types.
Let's rewrite our pair class as a class template (using our class template in a function):

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
constexpr T max(Pair<T> p)
{
    return (p.first < p.second ? p.second : p.first);
}

int main()
{
    Pair<int> p1{ 5, 6 };
    std::cout << max<int>(p1) << " is larger\n"; // explicit call to
        max<int>

    Pair<double> p2{ 1.2, 3.4 };
    std::cout << max(p2) << " is larger\n"; // call to max<double> using
        template argument deduction (prefer this)

    return 0;
}
```

Class templates with template type and non-template type members

Class templates can have some members using a template type and other members using a normal (non-template) type. For example:

```
template <typename T>
struct Foo
{
    T first{};    // first will have whatever type T is replaced with
    int second{}; // second will always have type int, regardless of what
        type T is
};
```

Same can be done with multiple template types.

13.5.1 std::pair

Because working with pairs of data is common, the C++ standard library contains a class template named std::pair (in the ¡utility¿ header) that is defined identically to the Pair class template with multiple template types in the preceding section. In fact, we can swap out the pair struct we developed for `std::pair`:

```
#include <iostream>
#include <utility>

template <typename T, typename U>
void print(std::pair<T, U> p)
{
```

```cpp
    // the members of std::pair have predefined names `first` and `second`
    std::cout << '[' << p.first << ", " << p.second << ']';
}

int main()
{
    std::pair<int, double> p1{ 1, 2.3 }; // a pair holding an int and a
        double
    std::pair<double, int> p2{ 4.5, 6 }; // a pair holding a double and an
        int
    std::pair<int, int> p3{ 7, 8 };      // a pair holding two ints

    print(p2);

    return 0;
}
```

13.5.2 Using class templates in multiple files

Just like function templates, class templates are typically defined in header files so they can be included into any code file that needs them. Both template definitions and type definitions are exempt from the one-definition rule, so this won't cause problems:
pair.h:

```cpp
#ifndef PAIR_H
#define PAIR_H

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
constexpr T max(Pair<T> p)
{
    return (p.first < p.second ? p.second : p.first);
}

#endif
```

foo.cpp:

```cpp
#include "pair.h"
#include <iostream>

void foo()
{
    Pair<int> p1{ 1, 2 };
    std::cout << max(p1) << " is larger\n";
}
```

main.cpp:

```cpp
#include "pair.h"
#include <iostream>

void foo(); // forward declaration for function foo()

int main()
{
    Pair<double> p2 { 3.4, 5.6 };
    std::cout << max(p2) << " is larger\n";

    foo();

    return 0;
}
```

Chapter 14

Classes

14.1 Introduction to object-oriented programming

Up to now, we've been doing a type of programming called procedural programming. In **procedural programming**, the focus is on creating "procedures" (which in C++ are called functions) that implement our program logic. We pass data objects to these functions, those functions perform operations on the data, and then potentially return a result to be used by the caller.
In programming, properties are represented by objects, and behaviors are represented by functions. And thus, procedural programming represents reality fairly poorly, as it separates properties (objects) and behaviors (functions).
In **object-oriented programming** (often abbreviated as OOP), the focus is on creating program-defined data types that contain both properties and a set of well-defined behaviors. The term "object" in OOP refers to the objects that we can instantiate from such types.
Because the properties and behaviors are no longer separate, objects are easier to modularize, which makes our programs easier to write and understand, and also provides a higher degree of code reusability. These objects also provide a more intuitive way to work with our data by allowing us to define how we interact with the objects, and how they interact with other objects.

14.2 Introduction to classes

14.2.1 The class invariant problem

Perhaps the biggest difficulty with structs is that they do not provide an effective way to document and enforce class invariants. An **invariant** is "a condition that must be true while some component is executing".
In the context of class types (which include structs, classes, and unions), a class invariant is a condition that must be true throughout the lifetime of an object in order for the object to remain in a valid state. An object that has a violated class invariant is said to be in an invalid state, and unexpected or undefined behavior may result from further use of that object.
Consider the following struct:

```
struct Fraction
{
    int numerator { 0 };
    int denominator { 1 };
};
```

We know from mathematics that a fraction with a denominator of 0 is mathematically undefined (because the value of a fraction is its numerator divided by its denominator – and division by 0 is mathematically undefined). Therefore, we want to ensure the denominator member of a Fraction object is never set to 0. If it is, then that Fraction object is in an invalid state, and undefined behavior may result from further use of that object.
For example:

```
#include <iostream>

struct Fraction
{
    int numerator { 0 };
    int denominator { 1 }; // class invariant: should never be 0
};

void printFractionValue(const Fraction& f)
{
     std::cout << f.numerator / f.denominator << '\n';
}

int main()
{
    Fraction f { 5, 0 };    // create a Fraction with a zero denominator
    printFractionValue(f); // cause divide by zero error

    return 0;
}
```

In the above example, we use a comment to document the invariant of Fraction. We also provide a default member initializer to ensure that denominator is set to 1 if the user does not provide an initialization value. This ensures our Fraction object will be valid if the user decides to value initialize a Fraction object. That's an okay start.
But nothing prevents us from explicitly violating this class invariant: When we create Fraction f, we use aggregate initialization to explicitly initialize the denominator to 0. While this does not cause an immediate issue, our object is now in an invalid state, and further use of the object may cause unexpected or undefined behaviour.

14.2.2 Definition

Just like structs, a **class** is a program-defined compound type that can have many member variables with different types.
Because a class is a program-defined data type, it must be defined before it can be used. Classes are defined similarly to structs, except we use the class keyword instead of struct. For example, here is a definition for a simple employee class:

```
        class Employee
{
    int m_id {};
    int m_age {};
    double m_wage {};
};
```

To demonstrate how similar classes and structs can be, consider that the following programs are completely equivalent to one another:

```
struct Date
{
    int m_day{};
    int m_month{};
    int m_year{};
};

void printDate(const Date& date)
{
    std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
}
```

```
class Date        // we changed struct to class
{
public:           // and added this line, which is called an access
    specifier
    int m_day{}; // and added "m_" prefixes to each of the member names
    int m_month{};
    int m_year{};
};

void printDate(const Date& date)
```

```cpp
{
    std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
}
```

Note: Most of the C++ standard library is classes
You have already been using class objects, perhaps without knowing it. Both std::string and std::string_view are defined as classes. In fact, most of the non-aliased types in the standard library are defined as classes!
Classes are really the heart and soul of C++ – they are so foundational that C++ was originally named "C with classes"! Once you are familiar with classes, much of your time in C++ will be spent writing, testing, and using them.

14.3 Member Functions

https://www.learncpp.com/cpp-tutorial/member-functions/
In addition to having member variables, class types (which includes structs, classes, and unions) can also have their own functions! Functions that belong to a class type are called **member functions**.
Functions that belong to a class type are called member functions.
Member functions must be declared inside the class type definition, and can be defined inside or outside of the class type definition.

```cpp
struct Date
{
    int year {};
    int month {};
    int day {};

    void print() // defines a member function named print
    {
        std::cout << year << '/' << month << '/' << day;
    }
};
```

Note: In the member function case, we don't need to pass today as an argument. The object that a member function is called on is implicitly passed to the member function. For this reason, the object that a member function is called on is often called the **implicit object**.

14.3.1 Accessing members

Accessing members inside a member function uses the implicit object

```cpp
void print() // defines a member function named print()
{
    std::cout << year << '/' << month << '/' << day;
}
```

In the member example, we access the members as year, month, and day.
Inside a member function, any member identifier that is not prefixed with the member selection operator (.) is associated with the implicit object.
For example:

```cpp
// Member function version
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() // defines a member function named print
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

```

```cpp
int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    today.day = 16; // member variables accessed using member selection
        operator (.)
    today.print();  // member functions also accessed using member
        selection operator (.)

    return 0;
}
```

When today.print() is called, today is our implicit object, and year, month, and day (which are not prefixed) evaluate to the values of today.year, today.month, and today.day respectively.

Member variables and functions can be defined in any order

Member functions defined inside the class type definition are implicitly inline, so they will not cause violations of the one-definition rule if the class type definition is included into multiple code files.

Class types with no data members

It is possible to create class types with no data members (e.g. class types that only have member functions). It is also possible to instantiate objects of such a class type:

```cpp
struct Foo
{
    void printHi() { std::cout << "Hi!\n"; }
};
```

However, if a class type does not have any data members, then using a class type is probably overkill. In such cases, consider using a namespace (containing non-member functions) instead.

14.4 Const member munctions

`https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/`

Const class objects

```cpp
    const Date today { 2020, 10, 14 }; // const class type object
```

Modifying the data members of const objects is disallowed.
Const objects may not call non-const member functions even if the function doesn't modify any member variable.
You may be surprised to find that this code also causes a compilation error:

```cpp
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print()
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.print();  // compile error: can't call non-const member function

    return 0;
}
```

Const member functions

A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions. Const member functions may also be called on non-const objects.

```cpp
struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // now a const member function
    {
        std::cout << year << '/' << month << '/' << day;
    }
};
```

Best practice: A member function that does not (and will not ever) modify the state of the object should be made const, so that it can be called on both const and non-const objects.

Finally, although it is not done very often, it is possible to overload a member function to have a const and non-const version of the same function.

```cpp
struct Something
{
    void print()
    {
        std::cout << "non-const\n";
    }

    void print() const
    {
        std::cout << "const\n";
    }
};
```

14.5 Member access

`https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/`
Each member of a class type has a property called an **access level** that determines who can access that member.
C++ has three different access levels: public, private, and protected. In this lesson, we'll cover the two commonly used access levels: public and private.
Whenever a member is accessed, the compiler checks whether the access level of the member permits that member to be accessed. If the access is not permitted, the compiler will generate a compilation error.

Public members: Members of a class type that do not have any restrictions on how they can be accessed. By default, all members of a struct are public members.

Private members: Members of a class type that can only be accessed by other members of the same class. By default, all members of a class are private members.

14.5.1 Default access level

Members that have the public access level are called public members. **Public members** are members of a class type that do not have any restrictions on how they can be accessed. Much like the park in our opening analogy, public members can be accessed by anyone (as long as they are in scope).
Members that have the private access level are called private members. **Private members** are members of a class type that can only be accessed by other members of the same class.

- By default, all members of a **struct** are **public members**.
- By default, all members of a **class** are **private members**.

```
struct Date
{
    // struct members are public by default, can be accessed by anyone
    int year {};        // public by default
    int month {};       // public by default
    int day {};         // public by default

    void print() const // public by default
    {
        // public members can be accessed in member functions of the class
            type
        std::cout << year << '/' << month << '/' << day;
    }
};
```

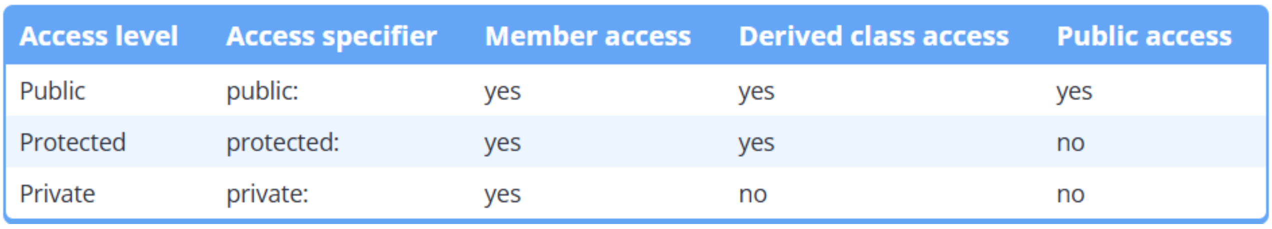
```
class Date // now a class instead of a struct
{
    // class members are private by default, can only be accessed by other
        members
    int m_year {};      // private by default
    int m_month {};     // private by default
    int m_day {};       // private by default

    void print() const // private by default
    {
        // private members can be accessed in member functions
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }
};
```

However, we can explicitly set the access level of our members by using an access specifier.

```
struct Date
{
// Any members defined here would default to public

private: // here's our private access specifier
    ...
public: // here's our public access specifier
    ...
};
```

```
class Date
{
// Any members defined here would default to private

public: // here's our public access specifier
    ...
private: // here's our private access specifier
    ...
};
```

Access level summary



Access level	Access specifier	Member access	Derived class access	Public access
Public	public:	yes	yes	yes
Protected	protected:	yes	yes	no
Private	private:	yes	no	no

14.5.2 Naming your private member variables

In C++, it is a common convention to name private data members starting with an "m_" prefix.
Consider the following member function of some class:

```cpp
// Some member function that sets private member m_name to the value of
    the name parameter
void setName(std::string_view name)
{
    m_name = name;
}
```

First, the "m_" prefix allows us to easily differentiate data members from function parameters or local variables within a member function. We can easily see that "m_name" is a member, and "name" is not. This helps make it clear that this function is changing the state of the class. And that is important because when we change the value of a data member, it persists beyond the scope of the member function (whereas changes to function parameters or local variables typically do not).

Best Practice: Consider naming your private data members starting with an "m_" prefix to help distinguish them from the names of local variables, function parameters, and member functions.

Public members of classes may also follow this convention if desired.

Another possible convention is just to add an underscore at the end of the member name: `position_`. Preferred by Francesco Giacomini.

14.5.3 Access level best practices for structs and classes

Now that we've covered what access levels are, let's talk about how we should use them.
Structs should avoid access specifiers altogether, meaning all struct members will be public by default. We want our structs to be aggregates, and aggregates can only have public members. Using the public: access specifier would be redundant with the default, and using private: or protected: would make the struct a non-aggregate.
Classes should generally only have private (or protected) data members (either by using the default private access level, or the private: (or protected:) access specifier).
Classes normally have public member functions (so those member functions can be used by the public after the object is created). However, occasionally member functions are made private (or protected) if they are not intended to be used by the public.

Best Practice: Classes should generally make member variables private (or protected), and member functions public.

Structs should generally avoid using access specifiers (all members will default to public).

14.6 Access functions

`https://www.learncpp.com/cpp-tutorial/access-functions/`
An access function is a trivial public member function whose job is to retrieve or change the value of a private member variable.

Getters: Public member functions that return the value of a private member variable. Getters are usually made const.

Setters: Public member functions that set the value of a private member variable.

Getters should provide "read-only" access to data. Therefore, the best practice is that they should return by either value (if making a copy of the member is inexpensive) or by const lvalue reference (if making a copy of the member is expensive). Getters are usually made const, so they can be called on both const and non-const objects. Setters should be non-const, so they can modify the data members.

```cpp
#include <iostream>

class Date
{
private:
    int m_year { 2020 };
    int m_month { 10 };
    int m_day { 14 };

public:
    void print()
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }

    int getYear() const { return m_year; }         // getter for year
    void setYear(int year) { m_year = year; }      // setter for year

    int getMonth() const  { return m_month; }      // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() const { return m_day; }           // getter for day
    void setDay(int day) { m_day = day; }          // setter for day
};

int main()
{
    Date d{};
    d.setYear(2021);
    std::cout << "The year is: " << d.getYear() << '\n';

    return 0;
}
```

Access function naming

```cpp
int getDay() const { return m_day; }  // getter
void setDay(int day) { m_day = day; } // setter
```

14.7 Member functions returning references

https://www.learncpp.com/cpp-tutorial/member-functions-returning-references/
Member functions can also return data members by (const) lvalue reference.
Data members have the same lifetime as the object containing them. Since member functions are always called on an object, and that object must exist in the scope of the caller, it is generally safe for a member function to return a data member by (const) lvalue reference (as the member being returned by reference will still exist in the scope of the caller when the function returns).
Returning by reference to the caller, we avoid having to make an expensive copy.

```
const std::string& getName() const { return m_name; } //  getter returns
    by const reference
```

Here's an example where it could be useful to return by const reference:

```
#include <iostream>
#include <string>

class Employee
{
  std::string m_name{};

public:
  void setName(std::string_view name) { m_name = name; }
  const auto& getName() const { return m_name; } // uses 'auto' to deduce
      return type from m_name
};

int main()
{
  Employee joe{}; // joe exists until end of function
  joe.setName("Joe");

  std::cout << joe.getName(); // returns joe.m_name by reference

  return 0;
}
```

Note: Prefer to use the return value of a member function that returns by reference immediately, to avoid issues with dangling references when the implicit object is an rvalue.

Non-const references to private data members

Because a reference acts just like the object being referenced, a member function that returns a non-const reference provides direct access to that member

Warning: Do not return non-const references to private data members

14.8 The benefits of data hiding (encapsulation)

https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
Class interface: Defines how a user of the class type will interact with objects of the class type. Because only public members can be accessed from outside of the class type, the public members of a class type form its interface.
Implementation: Consists of the code that actually makes the class behave as intended. This includes both the member variables that store data, and the bodies of the member functions that contain the program logic and manipulate the member variables.
Data hiding: A technique used to enforce the separation of interface and implementation by hiding (making inaccessible) the implementation of a program-defined data type from users.
The benefits of data hiding are the following:

- Making classes easier to use, and reducing complexity
- Allowing to maintain invariants
- Allowing to do better error detection
- Making it possible to change implementation details without breaking existing programs
- Classes with interfaces are easier to debug

Making classes easier to use, and reducing complexity

[...]

Allowing to maintain invariants

[...]

Allowing to do better error detection

[...]

Making it possible to change implementation details without breaking existing programs

[...]

Classes with interfaces are easier to debug

[...]

14.8.1 Prefer non-member functions to member functions

In C++, if a function can be reasonably implemented as a non-member function, prefer to implement it as a non-member function instead of as a member function.

- Non-member functions are not part of the interface of your class. Thus, the interface of your class will be smaller and more straightforward, making the class easier to understand.
- Non-member functions enforce encapsulation, as such functions must work through the public interface of the class. There is no temptation to access the implementation directly just because it is convenient.
- Non-member functions do not need to be considered when making changes to the implementation of a class (so long as the interface doesn't change in an incompatible way).
- Non-member functions tend to be easier to debug.
- Non-member functions containing application specific data and logic can be separated from the reusable portions of the class.

Best practice: Prefer implementing functions as non-members when possible (especially functions that contain application specific data or logic).

14.8.2 The order of class member declaration

When writing code outside of a class, we are required to declare variables and functions before we can use them. However, inside a class, this limitation does not exist. As noted in lesson 14.3 – Member functions, we can order our members in any order we like.
So how should we order them?
There are two schools of thought here:

- List your private members first, and then list your public member functions. This follows the traditional style of declare-before-use. Anybody looking at your class code will see how you've defined your data members before they are used, which can make reading through and understanding implementation details easier.
- List your public members first, and put your private members down at the bottom. Because someone who uses your class is interested in the public interface, putting your public members first makes the information they need up top, and puts the implementation details (which are least important) last.

In modern C++, the second method (public members go first) is more commonly recommended, especially for code that will be shared with other developers.

14.9 Constructors

`https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/`

14.9.1 Aggregate initialization

:

```cpp
struct Foo // Foo is an aggregate
{
    int x {};
    int y {};
};

int main()
{
    Foo foo { 6, 7 }; // uses aggregate initialization

    return 0;
}
```

However, as soon as we make any member variables private we're no longer able to use aggregate initialization.

```cpp
class Foo // Foo is not an aggregate (has private members)
{
    int m_x {};
    int m_y {};
};

int main()
{
    Foo foo { 6, 7 }; // compile error: can not use aggregate
        initialization

    return 0;
}
```

Not allowing class types with private members to be initialized via aggregate initialization makes sense for a number of reasons:

- Aggregate initialization requires knowing about the implementation of the class (since you have to know what the members are, and what order they were defined in), which we're intentionally trying to avoid when we hide our data members.
- If our class had some kind of invariant, we'd be relying on the user to initialize the class in a way that preserves the invariant.

Constructor: A special member function that is automatically called after a non-aggregate class type object is created.

When a non-aggregate class type object is defined, the compiler looks to see if it can find an accessible constructor that is a match for the initialization values provided by the caller.

- If an accessible matching constructor is found, memory for the object is allocated, and then the constructor function is called.
- If no accessible matching constructor can be found, a compilation error will be generated.

Beyond determining how an object may be created, constructors generally perform two functions:

- They typically perform initialization of any member variables (via a member initialization list)
- They may perform other setup functions (via statements in the body of the constructor).

Note: Aggregates are not allowed to have constructors so if you add a constructor to an aggregate, it is no longer an aggregate.

14.9.2 Naming constructors

: Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class, with the same capitalization. (For template classes, this name excludes the template parameters).
- Constructors have no return type (not even void).

Because constructors are typically part of the interface for your class, they are usually public. A constructor needs to be able to initialize the object being constructed, therefore, it must not be const.
Example:

```cpp
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y) // here's our constructor function that takes two
        initializers
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 }; // calls Foo(int, int) constructor
    foo.print();

    return 0;
}
```

Constructors and setters: Constructors are designed to initialize an entire object at the point of instantiation. Setters are designed to assign a value to a single member of an existing object.

14.10 Constructor member initializer lists

`https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/`

14.10.1 member initialization list

Member initialization via a member initialization list
To have a constructor initialize members, we do so using a member initializer list (often called a “member initialization list”). Do not confuse this with the similarly named “initializer list” that is used to initialize aggregates with a list of values.
Member initialization lists are something that is best learned by example. In the following example, our Foo(int, int) constructor has been updated to use a member initializer list to initialize m_x, and m_y:

```cpp
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y)
        : m_x { x }, m_y { y } // here's our member initialization list
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 };
    foo.print();

    return 0;
}
```

The member initializer list is defined after the constructor parameters. It begins with a colon (:), and then lists each member to initialize along with the initialization value for that variable, separated by a comma. You must use a direct form of initialization here (preferably using braces, but parentheses works as well) – using copy initialization (with an equals) does not work here. Also note that the member initializer list does not end in a semicolon.
This program produces the following output:

```
Foo(6, 7) constructed
Foo(6, 7)
```

Because the C++ standard says so, the members in a member initializer list are always initialized in the order in which they are defined inside the class
To help prevent such errors, members in the member initializer list should be listed in the order in which they are defined in the class. Some compilers will issue a warning if members are initialized out of order.
The bodies of constructors functions are most often left empty. This is because we primarily use constructor for initialization, which is done via the member initializer list. However, because the statements in the body of the constructor execute after the member initializer list has executed, we can add statements to do any other setup tasks required.

Member initializer list vs default member initializers

Members can be initialized in a few different ways:

- If a member is listed in the member initializer list, that initialization value is used
- Otherwise, if the member has a default member initializer, that initialization value is used
- Otherwise, the member is default-initialized.

14.10.2 Constructor function bodies

New programmers sometimes use the body of the constructor to assign values to members:

```
#include <iostream>

class Foo
{
private:
    int m_x { 0 };
    int m_y { 1 };

public:
    Foo(int x, int y)
    {
        m_x = x; // incorrect: this is an assignment, not an initialization
        m_y = y; // incorrect: this is an assignment, not an initialization
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo { 6, 7 };
    foo.print();

    return 0;
}
```

Although in this simple case this will produce the expected result, in case where members are required to be initialized (such as for data members that are const or references) assignment will not work.

Detecting and handling invalid arguments to constructors

Consider the following Fraction class:

```
class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    Fraction(int numerator, int denominator):
        m_numerator { numerator }, m_denominator { denominator }
    {
    }
};
```

Inside a member initializer list, our tools for detecting and handling errors are quite limited. We can use the conditional operator to detect an error, but then what?

```
class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    Fraction(int numerator, int denominator):
        m_numerator { numerator }, m_denominator { denominator != 0.0 ?
            denominator : ??? } // what do we do here?
    {
    }
};
```

We could change the denominator to a valid value, but then the user is going to get a Fraction that doesn't contain the values they asked for, and we don't have any way to notify them that we did something unexpected. Thus, we typically won't try to do any kind of validation in the member initializer list – we'll just initialize the members with the values passed in, and then try to deal with the situation.
Inside the body of the constructor, we can use statements, so we have more options for detecting and handling errors. This is a good place to assert or `static_assert` that the arguments passed in are semantically valid, but that doesn't actually handle runtime errors in a production build.
When a constructor cannot construct a semantically valid object, we say it has failed.

When constructors fail

Throw an exception: exceptions abort the construction process entirely, which means the user never gets access to a semantically invalid object. So in most cases, throwing an exception is the best thing to do in these situations.

14.11 Default constructors

https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/
A **default constructor** is a constructor that accepts no arguments. Typically, this is a constructor that has been defined with no parameters.

```cpp
public:
    Foo() // default constructor
    {
        std::cout << "Foo default constructed\n";
    }
```

When the above program runs, an object of type Foo is created. Since no initialization values have been provided, the default constructor Foo() is called, which prints:

```
Foo default constructed
```

14.11.1 Value initialization vs default initialization for class types

If a class type has a default constructor, both value initialization and default initialization will call the default constructor.

```cpp
Foo foo{}; // value initialization, calls Foo() default constructor
Foo foo2;  // default initialization, calls Foo() default constructor
```

However, value initialization is safer for aggregates. Since it's difficult to tell whether a class type is an aggregate or non-aggregate, it's safer to just use value initialization for everything and not worry about it.

14.11.2 Constructors with default arguments

If all of the parameters in a constructor have default arguments, the constructor is a default constructor (because it can be called with no arguments).

```cpp
public:
    Foo(int x=0, int y=0) // has default arguments
        : m_x { x }
        , m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
```

14.11.3 Overloaded constructors

Because constructors are functions, they can be overloaded. That is, we can have multiple constructors so that we can construct objects in different ways:

```cpp
public:
    Foo() // default constructor
    {
        std::cout << "Foo constructed\n";
    }

    Foo(int x, int y) // non-default constructor
        : m_x { x }, m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
```

A corollary of the above is that a class should only have one default constructor. If more than one default constructor is provided, the compiler will be unable to disambiguate which should be used.
If a non-aggregate class type object has no user-declared constructors, the compiler will generate a public default constructor. This constructor is called an **implicit default constructor**.
Generate an explicitly defaulted default constructor:

```
Foo() = default; // generates an explicitly defaulted default constructor
```

Only create a default constructor when it makes sense.

14.12 Delegating constructors

https://www.learncpp.com/cpp-tutorial/delegating-constructors/
Constructors are allowed to delegate initialization to another constructor of the same class type.

```
    public:
    Employee(std::string_view name)
        : Employee{ name, 0 } // delegate initialization to
          Employee(std::string_view, int) constructor
    {
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id{ id } // actually initializes the members
    {
        std::cout << "Employee " << m_name << " created\n";
    }
```

Reducing constructors using default arguments:
Default values can also sometimes be used to reduce multiple constructors into fewer constructors. For example, by putting a default value on our id parameter, we can create a single Employee constructor that requires a name argument but will optionally accept an id argument:

```
    public:

    Employee(std::string_view name, int id = 0) // default argument for id
        : m_name{ name }, m_id{ id }
    {
        std::cout << "Employee " << m_name << " created\n";
    }
```

14.13 Temporary class objects

https://www.learncpp.com/cpp-tutorial/temporary-class-objects/
In most cases where a variable is used only once, we actually don't need a variable. Instead, we can substitute in the expression used to initialize the variable where the variable would have been used.

```
    int main()
{
    // Case 1: Pass variable
    IntPair p { 3, 4 };
    print(p);

    // Case 2: Construct temporary IntPair and pass to function
    print(IntPair { 5, 6 } );

    // Case 3: Implicitly convert { 7, 8 } to a temporary Intpair and pass
       to function
    print( { 7, 8 } );

    return 0;
}
```

Temporary objects and return by value:

```
// Case 1: Create named variable and return
IntPair ret1()
{
    IntPair p { 3, 4 };
    return p; // returns temporary object (initialized using p)
}

// Case 2: Create temporary IntPair and return
IntPair ret2()
{
    return IntPair { 5, 6 }; // returns temporary object (initialized
        using another temporary object)
}

// Case 3: implicitly convert { 7, 8 } to IntPair and return
IntPair ret3()
{
    return { 7, 8 }; // returns temporary object (initialized using
        another temporary object)
}
```

14.14 Copy Constructor

https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/
A **copy constructor** is a constructor that is used to initialize an object with an existing object of the same type. After the copy constructor executes, the newly created object should be a copy of the object passed in as the initializer.

```
    Fraction f { 5, 3 };  // Calls Fraction(int, int) constructor
    Fraction fCopy { f }; // What constructor is used here?
```

If you do not provide a copy constructor for your classes, C++ will create a public **implicit copy constructor** for you. By default, the implicit copy constructor will do memberwise initialization.

14.14.1 Defining your own copy constructor

```
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
    {
    }

    // Copy constructor
    Fraction(const Fraction& fraction)
        // Initialize our members using the corresponding member of the
            parameter
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n"; // just to prove it works
    }
```

A copy constructor should not do anything other than copy an object. This is because the compiler may optimize the copy constructor out in certain cases. If you are relying on the copy constructor for some behavior other than just copying, that behavior may or may not occur.

Best practice: Prefer the implicit copy constructor.

Occasionally we run into cases where we do not want objects of a certain class to be copyable. We can prevent this by marking the copy constructor function as deleted, using the ”= delete” syntax:

```
public:
        // Delete the copy constructor so no copies can be made
    Fraction(const Fraction& fraction) = delete;
```

14.15 Class initialization and copy elision

`https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/`
All class initializations

```cpp
    // Calls Foo() default constructor
    Foo f1;           // default initialization
    Foo f2{};         // value initialization (preferred)

    // Calls foo(int) normal constructor
    Foo f3 = 3;       // copy initialization (non-explicit constructors
        only)
    Foo f4(4);        // direct initialization
    Foo f5{ 5 };      // direct list initialization (preferred)
    Foo f6 = { 6 };   // copy list initialization (non-explicit
        constructors only)

    // Calls foo(const Foo&) copy constructor
    Foo f7 = f3;      // copy initialization
    Foo f8(f3);       // direct initialization
    Foo f9{ f3 };     // direct list initialization (preferred)
    Foo f10 = { f3 }; // copy list initialization
```

Copy elision: Compiler optimization technique that allows the compiler to remove unnecessary copying of objects.
Copy elision is allowed to elide the copy constructor even if the copy constructor has side effects (such as printing text to the console)! This is why copy constructors should not have side effects other than copying

14.16 Converting constructors and the explicit keyword

`https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/`
A constructor that can be used to perform an implicit conversion is called a **converting constructor**. By default, all constructors are converting constructors.
There are some issues related to this we will not dive into.

14.16.1 The explicit keyword

To address issues, we can use the explicit keyword to tell the compiler that a constructor should not be used as a converting constructor.

```cpp
    public:
    explicit Dollars(int d) // now explicit
        : m_dollars{ d }
    {
    }
```

Making a constructor explicit has two notable consequences:

- An explicit constructor cannot be used to do copy initialization or copy list initialization.
- An explicit constructor cannot be used to do implicit conversions (since this uses copy initialization or copy list initialization).

```cpp
class Foo
{
public:
    explicit Foo() // note: explicit (just for sake of example)
    {
    }

    explicit Foo(int x) // note: explicit
    {
    }
};

```

```
Foo getFoo()
{
    // explicit Foo() cases
    return Foo{ };    // ok
    return { };       // error: can't implicitly convert initializer list
        to Foo

    // explicit Foo(int) cases
    return 5;         // error: can't implicitly convert int to Foo
    return Foo{ 5 }; // ok
    return { 5 };     // error: can't implicitly convert initializer list
        to Foo
}
```

An explicit constructor can still be used for direct and direct list initialization:

```
    // Assume Dollars(int) is explicit
int main()
{
    Dollars d1(5); // ok
    Dollars d2{5}; // ok
}
```

14.16.2 Best practices for use of explicit

The modern best practice is to make any constructor that will accept a single argument explicit by default. This includes constructors with multiple parameters where most or all of them have default values. This will disallow the compiler from using that constructor for implicit conversions. If an implicit conversion is required, only non-explicit constructors will be considered. If no non-explicit constructor can be found to perform the conversion, the compiler will error.
If such a conversion is actually desired in a particular case, it is trivial to convert the implicit conversion into an explicit definition using direct list initialization.
The following **should not be** made **explicit**:

- Copy (and move) constructors (as these do not perform conversions).

The following **are typically not** made **explicit**:

- Default constructors with no parameters (as these are only used to convert to a default object, not something we typically need to restrict).
- Constructors that only accept multiple arguments (as these are typically not a candidate for conversions anyway).

However, if you prefer, the above can be marked as explicit to prevent implicit conversions with empty and multiple-argument lists.
The following **should usually be** made **explicit**:

- Constructors that take a single argument.

Best practice: Make any constructor that accepts a single argument explicit by default. If an implicit conversion between types is both semantically equivalent and performant, you can consider making the constructor non-explicit.

Do not make copy or move constructors explicit, as these do not perform conversions.

14.17 Constexpr aggregates and classes

14.18 `this` pointer

Inside every member function, the keyword **this** is a const pointer that holds the address of the current implicit object.
Most of the time, we don't mention this explicitly, but just to prove we can:

```cpp
#include <iostream>

class Simple
{
private:
    int m_id{};

public:
    Simple(int id)
        : m_id{ id }
    {
    }

    int getID() const { return m_id; }
    void setID(int id) { m_id = id; }

    void print() const { std::cout << this->m_id; } // use `this` pointer
        to access the implicit object and operator-> to select member m_id
};

int main()
{
    Simple simple{ 1 };
    simple.setID(2);

    simple.print();

    return 0;
}
```

Consider this:

```cpp
void print() const { std::cout << m_id; }       // implicit use of this
void print() const { std::cout << this->m_id; } // explicit use of this
```

It turns out that the former is shorthand for the latter. When we compile our programs, the compiler will implicitly prefix any member referencing the implicit object with this-¿. This helps keep our code more concise and prevents the redundancy from having to explicitly write this-¿ over and over.

14.18.1 How is this set?

Let's take a closer look at this function call:

```cpp
simple.setID(2);
```

Although the call to function setID(2) looks like it only has one argument, it actually has two! When compiled, the compiler rewrites the expression simple.setID(2); as follows:

```cpp
Simple::setID(&simple, 2); // note that simple has been changed from an
    object prefix to a function argument!
```

Note that this is now just a standard function call, and the object simple (which was formerly an object prefix) is now passed by address as an argument to the function.
But that's only half of the answer. Since the function call now has an added argument, the member function definition also needs to be modified to accept (and use) this argument as a parameter. Here's our original member function definition for setID(). How the compiler rewrites functions is an implementation-specific detail, but the end-result is something like this:

```cpp
static void setID(Simple* const this, int id) { this->m_id = id; }
```

this always points to the object being operated on

New programmers are sometimes confused about how many this pointers exist. Each member function has a single this pointer parameter that points to the implicit object. Consider:

```
    Simple a{1}; // this = &a inside the Simple constructor
    Simple b{2}; // this = &b inside the Simple constructor
    a.setID(3); // this = &a inside member function setID()
    b.setID(4); // this = &b inside member function setID()
```

Note that the this pointer alternately holds the address of object a or b depending on whether we've called a member function on object a or b.
Because this is just a function parameter (and not a member), it does not make instances of your class larger memory-wise.

Explicitly referencing `this`

Most of the time, you won't need to explicitly reference the this pointer. However, there are a few occasions where doing so can be useful:
First, if you have a member function that has a parameter with the same name as a data member, you can disambiguate them by using this:

```
struct Something
{
    int data{}; // not using m_ prefix because this is a struct

    void setData(int data)
    {
        this->data = data; // this->data is the member, data is the local
            parameter
    }
};
```

Second, it can sometimes be useful to have a member function return the implicit object as a return value. The primary reason to do this is to allow member functions to be "chained" together, so several member functions can be called on the same object in a single expression! This is called function chaining (or method chaining). Consider the following class:

```
class Calc
{
private:
    int m_value{};

public:

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void mult(int value) { m_value *= value; }

    int getValue() const { return m_value; }
};
```

However, if we make each function return *this by reference, we can chain the calls together. Here is the new version of Calc with "chainable" functions:

```
class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }
};
```

Note that add(), sub() and mult() are now returning *this by reference. Consequently, this allows us to do the following:

```cpp
#include <iostream>

int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4); // method chaining

    std::cout << calc.getValue() << '\n';

    return 0;
}
```

We have effectively condensed three lines into one expression! Let's take a closer look at how this works.
Third, it can be useful to reset a class back to default state

```cpp
#include <iostream>

class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }

    void reset() { *this = {}; }
};


int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n'; // prints 8

    calc.reset();

    std::cout << calc.getValue() << '\n'; // prints 0

    return 0;
}
```

Note: For non-const member functions, this is a const pointer to a non-const value (meaning this cannot be pointed at something else, but the object pointing to may be modified).

With const member functions, this is a const pointer to a const value (meaning the pointer cannot be pointed at something else, nor may the object being pointed to be modified).

Why this a pointer and not a reference

Since the this pointer always points to the implicit object (and can never be a null pointer unless we've done something to cause undefined behaviour), you may be wondering why this is a pointer instead of a reference. The answer is simple: when this was added to C++, references didn't exist yet.
If this were added to the C++ language today, it would undoubtedly be a reference instead of a pointer. In other more modern C++-like languages, such as Java and C#, this is implemented as a reference.

14.19 Classes in header files

All of the classes that we have written so far have been simple enough that we have been able to implement the member functions directly inside the class definition itself.
However, as classes get longer and more complicated, having all the member function definitions inside the class can make the class harder to manage and work with. Using an already-written class only requires understanding its public interface (the public member functions), not how the class works underneath the hood. The member function implementations clutter up the public interface with details that aren't relevant to actually using the class.
To help address this, C++ allows us to separate the "declaration" portion of the class from the "implementation" portion by defining member functions outside of the class definition.
Consider the date class:

```cpp
#include <iostream>

class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    Date(int year, int month, int day)
        : m_year { year }
        , m_month { month }
        , m_day { day}
    {
    }

    void print() const { std::cout << "Date(" << m_year << ", " << m_month
        << ", " << m_day << ")\n"; }

    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};

int main()
{
    Date d { 2015, 10, 14 };
    d.print();

    return 0;
}
```

Here is the same Date class as above, with the constructor and print() member functions defined outside the class definition. Note that the prototypes for these member functions still exist inside the class definition (as these functions need to be declared as part of the class type definition), but the actual implementation has been moved outside:

```cpp
#include <iostream>

class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    Date(int year, int month, int day); // constructor declaration

    void print() const; // print function declaration

    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const  { return m_day; }
};

Date::Date(int year, int month, int day) // constructor definition
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

void Date::print() const // print function definition
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day <<
       ")\n";
};

int main()
{
    const Date d{ 2015, 10, 14 };
    d.print();

    return 0;
}
```

14.19.1 Class definitions in header files

Most often, classes are defined in header files of the same name as the class, and any member functions defined outside of the class are put in a .cpp file of the same name as the class.
Here's our Date class again, broken into a .cpp and .h file:
date.h

```cpp
#ifndef DATE_H
#define DATE_H

class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    Date(int year, int month, int day);

    void print() const;

    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};

#endif
```

date.cpp

```cpp
#include "Date.h"

Date::Date(int year, int month, int day) // constructor definition
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

void Date::print() const // print function definition
```

```
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day <<
        ")\n";
};
```

Now any other header or code file that wants to use the Date class can simply #include "date.h". Note that date.cpp also needs to be compiled into any project that uses date.h so that the linker can connect calls to the member functions to their definitions.

Best practice: Prefer to put your class definitions in a header file with the same name as the class. Trivial member functions (such as access functions, constructors with empty bodies, etc...) can be defined inside the class definition.

Prefer to define non-trivial member functions in a source file with the same name as the class.

You might be tempted to put all of your member function definitions into the header file, either inside the class definition, or as inline functions below the class definition. While this will compile, there are a couple of downsides to doing so.
First, as mentioned above, defining members inside the class definition clutters up your class definition.
Second, if you change any of the code in the header, then you'll need to recompile every file that includes that header. This can have a ripple effect, where one minor change causes the entire program to need to recompile. The cost of recompilation can vary significantly: a small project may only take a minute or less to build, whereas a large commercial project can take hours.
Conversely, if you change the code in a .cpp file, only that .cpp file needs to be recompiled. Therefore, given the choice, it's generally better to put non-trivial code in a .cpp file when you can.
There are a few cases where it might make sense to violate the best practice of putting the class definition in a header and non-trivial member functions in a code file.
First, for a small class that is used in only one code file and not intended for general reuse, you may prefer to define the class (and all member functions) directly in the single .cpp file it is used in. This helps make it clear that the class is only used within that single file, and is not intended for wider use. You can always move the class to a separate header/code file later if you later find you want to use it in more than one file, or are finding that the class and member function definitions are cluttering your source file.
Second, if a class only has a small number of non-trivial member functions that are unlikely to change, creating a .cpp file that contains only one or two definitions may not be worth the effort (as it clutters your project). In such cases, it may be preferable to make the member functions inline and place them beneath the class definition in the header.
Third, in modern C++, classes or libraries are increasingly being distributed as "header-only", meaning all of the code for the class or library is placed in a header file. This is done primarily to make distributing and using such files easier, as a header only needs to be #included, whereas a code file needs to be explicitly added to every project that uses it, so that it can be compiled. If intentionally creating a header-only class or library for distribution, all non-trivial member functions can be made inline and placed in the header file beneath the class definition.
Finally, for template classes, template member functions defined outside the class are almost always defined inside the header file, beneath the class definition. Just like non-member template functions, the compiler needs to see the full template definition in order to instantiate it. We cover these in a future lesson.

14.20 Nested types

14.21 Destructors

14.22 Class templates with member functions

14.23 Static member variables

14.24 Static member functions

14.25 Friend non-member functions

14.26 Friend classes and friend member functions

14.27 Ref qualifiers

Chapter 15

Operator Overloading

15.1 Operators as functions

`https://www.learncpp.com/cpp-tutorial/introduction-to-operator-overloading/`
n C++, operators are implemented as functions. By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you've written). Using function overloading to overload operators is called operator overloading.
In this chapter, we'll examine topics related to operator overloading.
When evaluating an expression containing an operator, the compiler uses the following rules:

- If all of the operands are fundamental data types, the compiler will call a built-in routine if one exists. If one does not exist, the compiler will produce a compiler error.
- If any of the operands are program-defined types (e.g. one of your classes, or an enum type), the compiler will use the function overload resolution algorithm (described in lesson 11.3 – Function overload resolution and ambiguous matches) to see if it can find an overloaded operator that is an unambiguous best match. This may involve implicitly converting one or more operands to match the parameter types of an overloaded operator. It may also involve implicitly converting program-defined types into fundamental types (via an overloaded typecast, which we'll cover later in this chapter) so that it can match a built-in operator. If no match can be found (or an ambiguous match is found), the compiler will error.

Limitations on operator overloading

First, almost any existing operator in C++ can be overloaded. The exceptions are: conditional (?:), sizeof, scope (::), member selector (.), pointer member selector (.*), typeid, and the casting operators.
Second, you can only overload the operators that exist. You can not create new operators or rename existing operators. For example, you could not create an operator** to do exponents.
Third, at least one of the operands in an overloaded operator must be a user-defined type. This means you could overload operator+(int, Mystring), but not operator+(int, double).
Fourth, it is not possible to change the number of operands an operator supports.
Finally, all operators keep their default precedence and associativity (regardless of what they're used for) and this can not be changed.

Best practice: When overloading operators, it's best to keep the function of the operators as close to the original intent of the operators as possible.

Best practice: If the meaning of an overloaded operator is not clear and intuitive, use a named function instead.

Best practice: Operators that do not modify their operands (e.g. arithmetic operators) should generally return results by value.

Operators that modify their leftmost operand (e.g. pre-increment, any of the assignment operators) should generally return the leftmost operand by reference.

15.2 Overloading using friend functions

https://www.learncpp.com/cpp-tutorial/overloading-operators-using-friend-functions/
The following example shows how to overload operator plus (+) in order to add two "Cents" objects together:

```
#include <iostream>

class Cents
{
private:
  int m_cents {};

public:
  Cents(int cents) : m_cents{ cents } { }

  // add Cents + Cents using a friend function
  friend Cents operator+(const Cents& c1, const Cents& c2);

  int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents& c1, const Cents& c2)
{
  // use the Cents constructor and operator+(int, int)
  // we can access m_cents directly because this is a friend function
  return c1.m_cents + c2.m_cents;
}

int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";
```

Overloading operators for operands of different types

Often it is the case that you want your overloaded operators to work with operands that are different types. For example, if we have Cents(4), we may want to add the integer 6 to this to produce the result Cents(10).
When C++ evaluates the expression x + y, x becomes the first parameter, and y becomes the second parameter. When x and y have the same type, it does not matter if you add x + y or y + x – either way, the same version of operator+ gets called. However, when the operands have different types, x + y does not call the same function as y + x.
For example, Cents(4) + 6 would call operator+(Cents, int), and 6 + Cents(4) would call operator+(int, Cents). Consequently, whenever we overload binary operators for operands of different types, we actually need to write two functions – one for each case. Here is an example of that:

```
#include <iostream>

class Cents
{
private:
  int m_cents {};

public:
  explicit Cents(int cents) : m_cents{ cents } { }

  // add Cents + int using a friend function
  friend Cents operator+(const Cents& c1, int value);

  // add int + Cents using a friend function
  friend Cents operator+(int value, const Cents& c1);


```

```cpp
    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents& c1, int value)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents { c1.m_cents + value };
}

// note: this function is not a member function!
Cents operator+(int value, const Cents& c1)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents { c1.m_cents + value };
}

int main()
{
    Cents c1{ Cents{ 4 } + 6 };
    Cents c2{ 6 + Cents{ 4 } };

    std::cout << "I have " << c1.getCents() << " cents.\n";
    std::cout << "I have " << c2.getCents() << " cents.\n";

    return 0;
}
```

15.3 Overloading operators using normal functions

https://www.learncpp.com/cpp-tutorial/overloading-operators-using-normal-functions/
Using a friend function to overload an operator is convenient because it gives you direct access to the internal members of the classes you're operating on. In the initial Cents example above, our friend function version of operator+ accessed member variable m_cents directly.
However, if you don't need that access, you can write your overloaded operators as normal functions. Note that the Cents class above contains an access function (getCents()) that allows us to get at m_cents without having to have direct access to private members. Because of this, we can write our overloaded operator+ as a non-friend:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
    Cents(int cents)
      : m_cents{ cents }
    {}

    int getCents() const { return m_cents; }
};

// note: this function is not a member function nor a friend function!
Cents operator+(const Cents& c1, const Cents& c2)
{
    // use the Cents constructor and operator+(int, int)
    // we don't need direct access to private members here
    return Cents{ c1.getCents() + c2.getCents() };
}

```

```cpp
int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

  return 0;
}
```

In general, a normal function should be preferred over a friend function if it's possible to do so with the existing member functions available (the less functions touching your classes's internals, the better). However, don't add additional access functions just to overload an operator as a normal function instead of a friend function!

15.4 Overloading the I/O operators

`https://www.learncpp.com/cpp-tutorial/overloading-the-io-operators/`

15.5 Overloading operators using member functions

`https://www.learncpp.com/cpp-tutorial/overloading-operators-using-member-functions/`
Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

- The overloaded operator must be added as a member function of the left operand.
- The left operand becomes the implicit *this object
- All other operands become function parameters.

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents {};

public:
    Cents(int cents)
        : m_cents { cents } { }

    // Overload Cents + int
    Cents operator+(int value) const;

    int getCents() const { return m_cents; }
};

// note: this function is a member function!
// the cents parameter in the friend version is now the implicit *this
    parameter
Cents Cents::operator+ (int value) const
{
    return Cents { m_cents + value };
}

int main()
{
  const Cents cents1 { 6 };
  const Cents cents2 { cents1 + 2 };
  std::cout << "I have " << cents2.getCents() << " cents.\n";

  return 0;
}
```

So if we can overload an operator as a friend or a member, which should we use? In order to answer that question, there's a few more things you'll need to know.

Not everything can be overloaded as a friend function

The assignment (=), subscript ([]), function call (()), and member selection (-¿) operators must be overloaded as member functions, because the language requires them to be.

Not everything can be overloaded as a member function

We are not able to overload operator¡¡ as a member function. Why not? Because the overloaded operator must be added as a member of the left operand. In this case, the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library. We can't modify the class declaration to add the overload as a member function of std::ostream.
Typically, we won't be able to use a member overload if the left operand is either not a class (e.g. int), or it is a class that we can't modify (e.g. std::ostream).

When to use a normal, friend, or member function overload

In most cases, the language leaves it up to you to determine whether you want to use the normal/friend or member function version of the overload. However, one of the two is usually a better choice than the other.
When dealing with binary operators that don't modify the left operand (e.g. operator+), the normal or friend function version is typically preferred, because it works for all parameter types (even when the left operand isn't a class object, or is a class that is not modifiable). The normal or friend function version has the added benefit of "symmetry", as all operands become explicit parameters (instead of the left operand becoming *this and the right operand becoming an explicit parameter).
When dealing with binary operators that do modify the left operand (e.g. operator+=), the member function version is typically preferred. In these cases, the leftmost operand will always be a class type, and having the object being modified become the one pointed to by *this is natural. Because the rightmost operand becomes an explicit parameter, there's no confusion over who is getting modified and who is getting evaluated.
Unary operators are usually overloaded as member functions as well, since the member version has no parameters.
The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (-¿), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand, but you can't add members to the class definition of the left operand (e.g. operator¡¡, which has a left operand of type ostream), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, do so as a member function.

15.6 Overloading unary operators (+, -)

`https://www.learncpp.com/cpp-tutorial/overloading-unary-operators/`

15.7 Overloading the comparison operators

`https://www.learncpp.com/cpp-tutorial/overloading-the-comparison-operators/`

15.8 Overloading the increment and decrement operators

`https://www.learncpp.com/cpp-tutorial/overloading-the-increment-and-decrement-operators/`

15.9 Overloading the subscript operator

`https://www.learncpp.com/cpp-tutorial/overloading-the-subscript-operator/`

15.10 Overloading typecasts

https://www.learncpp.com/cpp-tutorial/overloading-typecasts/

15.11 Overloading the assignment operator

https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/

Chapter 16

Arrays and vectors

16.1 Introduction to containers

Containers exist in programming, to make it easier to create and manage (potentially large) collections of objects. In general programming, a container is a data type that provides storage for a collection of unnamed objects (called elements).
As it turns out, you've already been using one container type: strings! A string container provides storage for a collection of characters, which can then be output as text:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name{ "Alex" }; // strings are a container for characters
    std::cout << name; // output our string as a sequence of characters

    return 0;
}
```

Note: The elements of a container are unnamed
While the container object itself typically has a name (otherwise how would we use it?), the elements of a container are unnamed. This is so that we can put as many elements in our container as we desire, without having to give each element a unique name! This lack of named elements is important, and is what distinguishes containers from other types of data structures.
But if the elements themselves are unnamed, how do we access them? Each container provides one or more methods to access its elements – but exactly how depends on the type of container. We'll see our first example of this in the next lesson.

16.1.1 The length of a container

In programming, the number of elements in a container is often called it's **length** (or sometimes count).
We already showed how we could use the length member function of std::string to get the number of character elements in the string container:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name{ "Alex" };
    std::cout << name << " has " << name.length() << " characters\n";

    return 0;
}
```

In C++, the term **size** is also commonly used for the number of elements in a container. This is an unfortunate choice of nomenclature, as the term "size" can also refer to the number of bytes of memory used by an object (as returned by the sizeof operator).

We'll prefer the term "length" when referring to the number of elements in a container, and use the term "size" to refer to amount of storage required by an object.

16.1.2 Container operations

Containers typically implement a significant subset of the following operations:

- Create a container (e.g. empty, with storage for some initial number of elements, from a list of values).
- Access to elements (e.g. get first element, get last element, get any element).
- Insert and remove elements.
- Get the number of elements in the container.

Containers may also provide other operations (or variations on the above) that assist in managing the collection of elements.
Modern programming languages typically provide a variety of different container types. These container types differ in terms of which operations they actually support, and how performant those operations are. For example, one container type might provide fast access to any element in the container, but not support insertion or removal of elements. Another container type might provide fast insertion and removal of elements, but only allow access to elements in sequential order.
Every container has a set of strengths and limitations. Picking the right container type for the task you are trying to solve can have a huge impact on both code maintainability and overall performance. We will discuss this topic further in a future lesson.

16.1.3 Element types

In most programming languages (including C++), containers are **homogenous**, meaning the elements of a container are required to have the same type.
Some containers use a preset element type (e.g. a string typically has char elements), but more often the element type can be set by the user of the container. In C++, containers are typically implemented as class templates, so that the user can provide the desired element type as a template type argument. We'll see an example of this next lesson.
This makes containers flexible, as we do not need to create a new container type for each different element type. Instead, we just instantiate the class template with the desired element type, and we're ready to go.
C++ contains three primary array types: (C-style) arrays, the std::vector container class, and the std::array container class.
Of the provided container classes, std::vector and std::array see by far the most use, and will be where we focus the bulk of our attention. The other containers classes are typically only used in more specialized situations.

Array: A container data type that stores a sequence of values contiguously (meaning each element is placed in an adjacent memory location, with no gaps). Arrays allow fast, direct access to any element. They are conceptually simple and easy to use, making them the first choice when we need to create and work with a set of related values.elated forward declarations into a code file.

16.2 std::vector

std::vector is one of the container classes in the C++ standard containers library that implements an array. std::vector is defined in the ¡vector¿ header as a class template, with a template type parameter that defines the type of the elements. Thus, std::vector¡int¿ declares a std::vector whose elements are of type int.
Instantiating a std::vector object is straightforward:

```cpp
#include <vector>

int main()
{
  // Value initialization (uses default constructor)
  std::vector<int> vec{}; // vector containing 0 int elements

  return 0;
}
```

Since the goal of a container is to manage a set of related values, most often we will want to initialize our container with those values. We can do this by using list initialization with the specific initialization values we want. For example:

```cpp
#include <vector>

int main()
{
  // List construction (uses list constructor)
  std::vector<int> primes{ 2, 3, 5, 7 };          // vector containing 4
      int elements with values 2, 3, 5, and 7
  std::vector vowels { 'a', 'e', 'i', 'o', 'u' }; // vector containing 5
      char elements with values 'a', 'e', 'i', 'o', and 'u'.  Uses CTAD
      (C++17) to deduce element type char (preferred).

  return 0;
}
```

16.2.1 List constructors and initializer lists

Containers typically have a special constructor called a list constructor that allows us to construct an instance of the container using an initializer list. The list constructor does three things:

- Ensures the container has enough storage to hold all the initialization values (if needed).
- Sets the length of the container to the number of elements in the initializer list (if needed).
- Initializes the elements to the values in the initializer list (in sequential order).

Thus, when we provide a container with an initializer list of values, the list constructor is called, and the container is constructed using that list of values!

16.2.2 Subscript operator

In C++, the most common way to access array elements is by using the name of the array along with the subscript operator (operator[]). To select a specific element, inside the square brackets of the subscript operator, we provide an integral value that identifies which element we want to select. This integral value is called a **subscript** (or informally, an **index**).
For example, primes[0] will return the element with index 0 (the first element) from the prime array. The subscript operator returns a reference to the actual element, not a copy. Once we've accessed an array element, we can use it just like a normal object (e.g. assign a value to it, output it, etc. . .)
Because the indexing starts with 0 rather than 1, we say arrays in C++ are zero-based. This can be confusing because we're used to counting objects starting from 1.
This also can cause some linguistic ambiguity, because when we talk about array element 1, it may not be clear whether we're talking about the first array element (with index 0) or the second array element (with index 1). Generally, we'll talk about array elements in terms of position rather than index (so the "first element" is the one with index 0).

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 }; // hold the first 5 prime
        numbers (as int)

    std::cout << "The first prime number is: " << primes[0] << '\n';
    std::cout << "The second prime number is: " << primes[1] << '\n';
    std::cout << "The sum of the first 5 primes is: " << primes[0] +
        primes[1] + primes[2] + primes[3] + primes[4] << '\n';

    return 0;
}
```

16.2.3 Subscript out of bounds

When indexing an array, the index provided must select a valid element of the array. That is, for an array of length N, the subscript must be a value between 0 and N-1 (inclusive).
operator[] does not do any kind of **bounds checking**, meaning it does not check to see whether the index is within the bounds of 0 to N-1 (inclusive). Passing an invalid index to operator[] will return in undefined behavior.

Arrays are contiguous in memory

One of the defining characteristics of arrays is that the elements are always allocated contiguously in memory, meaning the elements are all adjacent in memory (with no gaps between them).

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 }; // hold the first 5 prime
        numbers (as int)

    std::cout << "An int is " << sizeof(int) << " bytes\n";
    std::cout << &(primes[0]) << '\n';
    std::cout << &(primes[1]) << '\n';
    std::cout << &(primes[2]) << '\n';

    return 0;
}
```

On the author's machine, one run of the above program produced the following result:

```
An int is 4 bytes
00DBF720
00DBF724
00DBF728
```

You'll note that the memory addresses for these int elements are 4 bytes apart, the same as the size of an int on the author's machine.
This means arrays do not have any per-element overhead. It also allows the compiler to quickly calculate the address of any element in the array.
Arrays are one of the few container types that allow for **random access**, meaning any element in the container can be accessed directly (as opposed to sequential access, where elements must be accessed in a particular order). Random access to array elements is typically efficient, and makes arrays very easy to use. This is a primary reason why arrays are often preferred over other containers.

16.2.4 Constructing a std::vector of a specific length

Consider the case where we want the user to input 10 values that we'll store in a std::vector. In this case, we need a std::vector of length 10 before we have any values to put in the std::vector. How do we address this?
We could create a std::vector and initialize it with an initializer list with 10 placeholder values:

```cpp
std::vector<int> data { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; // vector
    containing 10 int values
```

But that's bad for a lot of reasons. It requires a lot of typing. It's not easy to see how many initializers there are. And it's not easy to update if we decide we want a different number of values later.
Fortunately, std::vector has an explicit constructor (explicit std::vector¡T¿(std::size_t)) that takes a single std::size_t value defining the length of the std::vector to construct:

```cpp
std::vector<int> data( 10 ); // vector containing 10 int elements,
    value-initialized to 0
```

Each of the created elements are value-initialized, which for int does zero-initialization (and for class types calls the default constructor).
However, there is one non-obvious thing about using this constructor: it must be called using direct initialization.

To help clarify what happens in various initialization cases further, let's look at similar cases using copy, direct, and list initialization:

```
// Copy init
std::vector<int> v1 = 10;     // 10 not an initializer list, copy init
    won't match explicit constructor: compilation error

// Direct init
std::vector<int> v2(10);      // 10 not an initializer list, matches
    explicit single-argument constructor

// List init
std::vector<int> v3{ 10 };    // { 10 } interpreted as initializer list,
    matches list constructor

// Copy list init
std::vector<int> v4 = { 10 }; // { 10 } interpreted as initializer list,
    matches list constructor
std::vector<int> v5({ 10 });  // { 10 } interpreted as initializer list,
    matches list constructor

        // Default init
        std::vector<int> v6 {};        // {} is empty initializer list,
            matches default constructor
        std::vector<int> v7 = {};      // {} is empty initializer list,
            matches default constructor
```

16.2.5 Const vectors

Objects of type std::vector can be made const:

```
#include <vector>

int main()
{
    const std::vector<int> prime { 2, 3, 5, 7, 11 }; // prime and its
        elements cannot be modified

    return 0;
}
```

A const std::vector must be initialized, and then cannot be modified. The elements of such a vector are treated as if they were const.
The element type of a std::vector must not be defined as const (e.g. std::vector¡const int¿ is disallowed).
One of the biggest downsides of std::vector is that it cannot be made constexpr. If you need a constexpr array, use std::array.

16.2.6 at() member function

The array container classes support another method for accessing an array. The at() member function can be used to do array access with runtime bounds checking:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector prime{ 2, 3, 5, 7, 11 };

    std::cout << prime.at(3); // print the value of element with index 3
    std::cout << prime.at(9); // invalid index (throws exception)

    return 0;
}
```

When the at() member function encounters an out-of-bounds index, it actually throws an exception of type std::out_of_range. If the exception is not handled, the program will be terminated. We cover exceptions and how to handle them in chapter 27.

16.2.7 Passing std::vector

An object of type std::vector can be passed to a function just like any other object. That means if we pass a std::vector by value, an expensive copy will be made. Therefore, we typically pass std::vector by (const) reference to avoid such copies.
With a std::vector, the element type is part of the type information of the object. Therefore, when we use a std::vector as a function parameter, we have to explicitly specify the element type:

```
void passByRef(const std::vector<int>& arr) // we must explicitly specify
    <int> here
{
    std::cout << arr[0] << '\n';
}
```

Because our passByRef() function expects a std::vector¡int¿, we are unable to pass vectors with different element types:

16.3 std::array

Dynamic arrays are powerful and convenient, but like everything in life, they make some tradeoffs for the benefits they offer.
std::vector is slightly less performant than the fixed-size arrays. In most cases you probably won't notice the difference (unless you're writing sloppy code that causes lots of inadvertent reallocations). std::vector only supports constexpr in very limited contexts. In modern C++, it is really this latter point that's significant. Constexpr arrays offer the ability to write code that is more robust, and can also be optimized more highly by the compiler. Whenever we can use a constexpr array, we should – and if we need a constexpr array, std::array is the container class we should be using.

16.3.1 Defining a std::array

```
#include <array>  // for std::array
#include <vector> // for std::vector

int main()
{
    std::array<int, 5> a {};  // a std::array of 5 ints

    std::vector<int> b(5);    // a std::vector of 5 ints (for comparison)

    return 0;
}
```

Our std::array declaration has two template arguments. The first (int) is a type template argument defining the type of the array element. The second (5) is an integral non-type template argument defining the array length.

The length of a std::array must be a constant expression

Unlike a std::vector, which can be resized at runtime, the length of a std::array must be a constant expression. Most often, the value provided for the length will be an integer literal, constexpr variable, or an unscoped enumerator.
Note: Non-const variables and runtime constants cannot be used for the length:

```
#include <array>
#include <iostream>

void foo(const int length) // length is a runtime constant
{
    std::array<int, length> e {}; // error: length is not a constant
        expression
```

```cpp
}

int main()
{
    // using a non-const variable
    int numStudents{};
    std::cin >> numStudents; // numStudents is non-constant

    std::array<int, numStudents> {}; // error: numStudents is not a
        constant expression

    foo(7);

    return 0;
}
```

Aggregate initialization of a std::array

Perhaps surprisingly, std::array is an aggregate. This means it has no constructors, and instead is initialized using aggregate initialization. As a quick recap, aggregate initialization allows us to directly initialize the members of aggregates. To do this, we provide an initializer list, which is a brace-enclosed list of comma-separated initialization values.

```cpp
#include <array>

int main()
{
    std::array<int, 6> fibonnaci = { 0, 1, 1, 2, 3, 5 }; // copy-list
        initialization using braced list
    std::array<int, 5> prime { 2, 3, 5, 7, 11 };          // list
        initialization using braced list (preferred)

    return 0;
}
```

Because we generally want our elements to be initialized, std::array should be value initialized (using empty braces) when defined with no initializers.

```cpp
    std::array<int, 5> a;   // Members default initialized (int elements
        are left uninitialized)
    std::array<int, 5> b{}; // Members value initialized (int elements are
        zero initialized) (preferred)

    std::vector<int> v(5);  // Members value initialized (int elements are
        zero initialized) (for comparison)
```

16.3.2 Const and constexpr std::array

16.3.3 Accessing array elements using operator

Just like a std::vector, the most common way to access elements of a std::array is by using the subscript operator (operator[]):

```cpp
#include <array> // for std::array
#include <iostream>

int main()
{
    constexpr std::array<int, 5> prime{ 2, 3, 5, 7, 11 };

    std::cout << prime[3]; // print the value of element with index 3 (7)
    std::cout << prime[9]; // invalid index (undefined behavior)

    return 0;
}
```

16.3.4 `std::array` length and indexing

The length of a std::array has type std::size_t

16.3.5 Getting the length of a std::array

16.3.6 Getting the length of a std::array as a constexpr value

std::get() does compile-time bounds checking for constexpr indices

16.3.7 Passing std::array

Using function templates to pass std::array of different element types or lengths

16.3.8 Returning std::array

16.4 C-style arrays

16.4.1 Declaring a C-style array

The array length of a c-style array must be a constant expression

16.4.2 Subscripting a C-style array

16.4.3 Aggregate initialization of C-style arrays

16.4.4 Const and constexpr C-style arrays

16.4.5 Getting the length of a C-style array

C-style arrays don't support assignment

16.4.6 C-style array decay

`https://www.learncpp.com/cpp-tutorial/c-style-array-decay/`
The designers of the C language had a problem. Consider the following simple program:

```cpp
#include <iostream>

void print(int val)
{
    std::cout << val;
}

int main()
{
    int x { 5 };
    print(x);

    return 0;
}
```

When print(x) is called, the value of argument x (5) is copied into parameter val. Within the body of the function, the value of val (5) is printed to the console. Because x is cheap to copy, there's no problem here.
Now consider the following similar program, which uses a 1000 element C-style int array instead of a single int:

```cpp
#include <iostream>

void printElementZero(int arr[1000])
{
    std::cout << arr[0]; // print the value of the first element
}

int main()
{
    int x[1000] { 5 };    // define an array with 1000 elements, x[0] is
        initialized to 5
    printElementZero(x);

    return 0;
}
```

This program also compiles and prints the expected value (5) to the console.
While the code in this example is similar to the code in the prior example, mechanically it works a bit different than you might expect (we'll explain this below). And that is due to the solution that the C designers came up to solve for two major challenges.
First, copying a 1000 element array every time a function is called is expensive (and even more so if the elements are an expensive-to-copy type), so we want to avoid that. But how? C doesn't have references, so using pass by reference to avoid making a copy of function arguments wasn't an option.
Second, we want to be able to write a single function that can accept array arguments of different lengths. Ideally, our printElementZero() function in the example above should be callable with arrays arguments of any length (since element 0 is guaranteed to exist). We don't want to have to write a different function for every possible array length that we want to use as an argument. But how? C has no syntax to specify "any length" arrays, nor does it support templates, nor can arrays of one length be converted to arrays of another length (presumably because doing so would involve making an expensive copy).
The designers of the C language came up with a clever solution (inherited by C++ for compatibility reasons) that solves for both of these issues:

```
#include <iostream>

void printElementZero(int arr[1000]) // doesn't make a copy
{
    std::cout << arr[0]; // print the value of the first element
}

int main()
{
    int x[7] { 5 };       // define an array with 7 elements
    printElementZero(x); // somehow works!

    return 0;
}
```

Somehow, the above example passes a 7 element array to a function expecting a 1000 element array, without any copies being made. In this lesson, we'll explore how this works.

Array to pointer conversions (array decay)

In most cases, when a C-style array is used in an expression, the array will be implicitly converted into a pointer to the element type, initialized with the address of the first element (with index 0). Colloquially, this is called **array decay** (or just **decay** for short).

```
#include <iomanip> // for std::boolalpha
#include <iostream>

int main()
{
    int arr[5]{ 9, 7, 5, 3, 1 }; // our array has elements of type int

    // First, let's prove that arr decays into an int* pointer

    auto ptr{ arr }; // evaluation causes arr to decay, type deduction
        should deduce type int*
    std::cout << std::boolalpha << (typeid(ptr) == typeid(int*)) << '\n';
        // Prints true if the type of ptr is int*

    // Now let's prove that the pointer holds the address of the first
        element of the array

    std::cout << std::boolalpha << (&arr[0] == ptr) << '\n';

    return 0;
}
```

On the author's machine, this printed:

```
true
true
```

Because C-style arrays decay into a pointer in most cases, it's a common fallacy to believe arrays are pointers. This is not the case. An array object is a sequence of elements, whereas a pointer object just holds an address. The type information of an array and a decayed array is different. In the example above, the array arr has type int[5], whereas the decayed array has type int*. Notably, the array type int[5] contains length information, whereas the decayed array pointer type int* does not.
Note: Subscripting a C-style array actually applies operator[] to the decayed pointer

```cpp
#include <iostream>

void printElementZero(const int* arr) // pass by const address
{
    std::cout << arr[0];
}

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 };
    const int squares[] { 1, 4, 9, 25, 36, 49, 64, 81 };

    printElementZero(prime);   // prime decays to an const int* pointer
    printElementZero(squares); // squares decays to an const int* pointer

    return 0;
}
```

One problem with declaring the function parameter as int* arr is that it's not obvious that arr is supposed to be a pointer to an array of values rather than a pointer to a single integer. For this reason, when passing a C-style array, its preferable to use the alternate declaration form int arr[]:

```cpp
void printElementZero(const int arr[]) // treated the same as const int*
{
    std::cout << arr[0];
}
```

This program behaves identically to the prior one, as the compiler will interpret function parameter const int arr[] the same as const int*. However, this has the advantage of communicating to the caller that arr is expected to be a decayed C-style array, not a pointer to a single value. Note that no length information is required between the square brackets (since it is not used anyway). If a length is provided, it will be ignored.

The problems with array decay

Although array decay was a clever solution to ensure C-style arrays of different lengths could be passed to a function without making expensive copies, the loss of array length information makes it easy for several types of mistakes to be made.
First, sizeof() will return different values for arrays and decayed arrays:

```cpp
#include <iostream>

void printArraySize(int arr[])
{
    std::cout << sizeof(arr) << '\n'; // prints 4 (assuming 32-bit
        addresses)
}

int main()
{
    int arr[]{ 3, 2, 1 };

    std::cout << sizeof(arr) << '\n'; // prints 12 (assuming 4 byte ints)

    printArraySize(arr);

    return 0;
}
```

Fortunately, C++17's better replacement std::size() (and C++20's std::ssize()) will fail to compile if passed a pointer value.
Second, and perhaps most importantly, array decay can make refactoring (breaking long functions into shorter, more modular functions) difficult. Code that works as expected with a non-decayed array may not compile (or worse, may silently malfunction) when the same code is using a decayed array.
Third, not having length information creates several programmatic challenges. Without length information, it is difficult to sanity check the length of the array. Users can easily pass in arrays that are shorter than expected (or even pointers to a single value), which will then cause undefined behavior when they are subscripted with an invalid index.

```
#include <iostream>

void printElement2(int arr[])
{
    // How do we ensure that arr has at least three elements?
    std::cout << arr[2] << '\n';
}

int main()
{
    int a[]{ 3, 2, 1 };
    printElement2(a);  // ok

    int b[]{ 7, 6 };
    printElement2(b);  // compiles but produces undefined behavior

    int c{ 9 };
    printElement2(&c); // compiles but produces undefined behavior

    return 0;
}
```

C-style arrays should be avoided in most cases

Best practice: Avoid C-style arrays whenever practical.

- Prefer std::string_view for read-only strings (string literal symbolic constants and string parameters).
- Prefer std::string for modifiable strings.
- Prefer std::array for non-global constexpr arrays.
- Prefer std::vector for non-constexpr arrays.
- It is okay to use C-style arrays for global constexpr arrays.

When are C-style arrays used in modern C++? In modern C++, C-style arrays are typically used in two cases:

- To store constexpr global (or constexpr static local) program data. Because such arrays can be accessed directly from anywhere in the program, we do not need to pass the array, which avoids decay-related issues. The syntax for defining C-style arrays can be a little less wonky than std::array. More importantly, indexing such arrays does not have sign conversion issues like the standard library container classes do.
- As parameters to functions or classes that want to handle non-constexpr C-style string arguments directly (rather than requiring a conversion to std::string_view). There are two possible reasons for this: First, converting a non-constexpr C-style string to a std::string_view requires traversing the C-style string to determine its length. If the function is in a performance critical section of code and the length isn't needed (e.g. because the function is going to traverse the string anyway) then avoiding the conversion may be useful. Second, if the function (or class) calls other functions that expect C-style strings, converting to

a std::string_view just to convert back may be suboptimal (unless you have other reasons for wanting a std::string_view).

16.5 Pointer arithmetic and subscripting

16.6 Multidimensional C-style Arrays

16.7 Multidimensional std::array

Chapter 17

Advanced Functions

17.1 Function Pointers

https://www.learncpp.com/cpp-tutorial/function-pointers/
A pointer is a variable that holds the address of another variable. Function pointers are similar, except that instead of pointing to variables, they point to functions!

17.1.1 Pointers to functions definition

The syntax for creating a non-const function pointer is one of the ugliest things you will ever see in C++:

```cpp
// fcnPtr is a pointer to a function that takes no arguments and returns
    an integer
int (*fcnPtr)();
```

In the above snippet, fcnPtr is a pointer to a function that has no parameters and returns an integer. fcnPtr can point to any function that matches this type.
The parentheses around *fcnPtr are necessary for precedence reasons, as int* fcnPtr() would be interpreted as a forward declaration for a function named fcnPtr that takes no parameters and returns a pointer to an integer.
To make a const function pointer, the const goes after the asterisk:

```cpp
int (*const fcnPtr)();
```

17.1.2 Assigning a function to a function pointer

Function pointers can be initialized with a function (and non-const function pointers can be assigned a function). Like with pointers to variables, we can also use &foo to get a function pointer to foo.

```cpp
int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    int (*fcnPtr)(){ &foo }; // fcnPtr points to function foo
    fcnPtr = &goo; // fcnPtr now points to function goo

    return 0;
}
```

One common mistake is to do this:

```cpp
fcnPtr = goo();
```

This tries to assign the return value from a call to function goo() (which has type int) to fcnPtr (which is expecting a value of type int(*)()), which isn't what we want. We want fcnPtr to be assigned the address of function goo, not the return value from function goo(). So no parentheses are needed.
Note that the type (parameters and return type) of the function pointer must match the type of the function. Here are some examples of this:

```
// function prototypes
int foo();
double goo();
int hoo(int x);

// function pointer initializers
int (*fcnPtr1)(){ &foo };     // okay
int (*fcnPtr2)(){ &goo };     // wrong -- return types don't match!
double (*fcnPtr4)(){ &goo }; // okay
fcnPtr1 = &hoo;               // wrong -- fcnPtr1 has no parameters, but
    hoo() does
int (*fcnPtr3)(int){ &hoo }; // okay
```

Unlike fundamental types, C++ will implicitly convert a function into a function pointer if needed (so you don't need to use the address-of operator (&) to get the function's address). However, function pointers will not convert to void pointers, or vice-versa (though some compilers like Visual Studio may allow this anyway).

```
// function prototypes
int foo();

// function initializations
int (*fcnPtr5)() { foo }; // okay, foo implicitly converts to function
    pointer to foo
void* vPtr { foo };        // not okay, though some compilers may allow
```

Function pointers can also be initialized or assigned the value nullptr:

```
int (*fcnptr)() { nullptr }; // okay
```

17.1.3 Calling a function using a function pointer

The other primary thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ &foo }; // Initialize fcnPtr with function foo
    (*fcnPtr)(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

The second way is via implicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ &foo }; // Initialize fcnPtr with function foo
    fcnPtr(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

As you can see, the implicit dereference method looks just like a normal function call – which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.
Also note that because function pointers can be set to nullptr, it's a good idea to assert or conditionally test whether your function pointer is a null pointer before calling it. Just like with normal pointers, dereferencing a null function pointer leads to undefined behavior.

17.1.4 Passing functions as arguments to other functions

One of the most useful things to do with function pointers is pass a function as an argument to another function. Functions used as arguments to another function are sometimes called **callback functions**.
Consider a case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.
Many comparison-based sorting algorithms work on a similar concept: the sorting algorithm iterates through a list of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison, we can change the way the algorithm sorts without affecting the rest of the sorting code.
Here is our selection sort routine from a previous lesson:

```
#include <utility> // for std::swap

void SelectionSort(int* array, int size)
{
    if (!array)
        return;

    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've
            encountered so far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at
            startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++
            currentIndex)
        {
            // If the current element is smaller than our previously found
                smallest
            if (array[smallestIndex] > array[currentIndex]) // COMPARISON
                DONE HERE
            {
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Let's replace that comparison with a function to do the comparison. Because our comparison function is going to compare two integers and return a boolean value to indicate whether the elements should be swapped, it will look something like this:

```
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}
```

And here's our selection sort routine using the ascending() function to do the comparison:

```
#include <utility> // for std::swap

void SelectionSort(int* array, int size)
{
    if (!array)
        return;

    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've
            encountered so far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at
            startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++
            currentIndex)
        {
            // If the current element is smaller than our previously found
                smallest
            if (ascending(array[smallestIndex], array[currentIndex])) //
                COMPARISON DONE HERE
            {
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Now, in order to let the caller decide how the sorting will be done, instead of using our own hard-coded comparison function, we'll allow the caller to provide their own sorting function! This is done via a function pointer.
Because the caller's comparison function is going to compare two integers and return a boolean value, a pointer to such a function would look something like this:

```
bool (*comparisonFcn)(int, int);
```

So, we'll allow the caller to pass our sort routine a pointer to their desired comparison function as the third parameter, and then we'll use the caller's function to do the comparison.
Here's a full example of a selection sort that uses a function pointer parameter to do a user-defined comparison, along with an example of how to call it:

```
#include <utility> // for std::swap
#include <iostream>

// Note our user-defined comparison is the third parameter
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int))
{
    if (!array || !comparisonFcn)
        return;

    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // bestIndex is the index of the smallest/largest element we've
            encountered so far.
        int bestIndex{ startIndex };

        // Look for smallest/largest element remaining in the array
            (starting at startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++
            currentIndex)
        {
            // If the current element is smaller/larger than our
                previously found smallest
            if (comparisonFcn(array[bestIndex], array[currentIndex])) //
                COMPARISON DONE HERE
            {
                // This is the new smallest/largest number for this
                    iteration
                bestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest/largest element
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// Here is a comparison function that sorts in ascending order
// (Note: it's exactly the same as the previous ascending() function)
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

// This function prints out the values in the array
void printArray(int* array, int size)
{
    if (!array)
        return;

    for (int index{ 0 }; index < size; ++index)
    {
        std::cout << array[index] << ' ';
    }

    std::cout << '\n';
}

int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
    printArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    printArray(array, 9);

    return 0;
}
```

This program produces the result:

```
9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9
```

Is that cool or what? We've given the caller the ability to control how our selection sort does its job.

If you're going to allow the caller to pass in a function as a parameter, it can often be useful to provide some standard functions for the caller to use for their convenience. For example, in the selection sort example above, providing the ascending() and descending() function along with the selectionSort() function would make the caller's life easier, as they wouldn't have to rewrite ascending() or descending() every time they want to use them.
You can even set one of these as a default parameter:

```
// Default the sort to ascending sort
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int) =
    ascending);
```

this case, as long as the user calls selectionSort normally (not through a function pointer), the comparisonFcn parameter will default to ascending. You will need to make sure that the ascending function is declared prior to this point, otherwise the compiler will complain it doesn't know what ascending is.

Making function pointers prettier with type aliases

Let's face it – the syntax for pointers to functions is ugly. However, type aliases can be used to make pointers to functions look more like regular variables:

```
using ValidateFunction = bool(*)(int, int);
```

This defines a type alias called "ValidateFunction" that is a pointer to a function that takes two ints and returns a bool.
Now instead of doing this:

```
bool validate(int x, int y, bool (*fcnPtr)(int, int)); // ugly
```

You can do this:

```
bool validate(int x, int y, ValidateFunction pfcn) // clean
```

17.1.5 Using `std::function`

An alternate method of defining and storing function pointers is to use std::function, which is part of the standard library ¡functional¿ header. To define a function pointer using this method, declare a std::function object like so:

```
#include <functional>
bool validate(int x, int y, std::function<bool(int, int)> fcn); //
    std::function method that returns a bool and takes two int parameters
```

As you see, both the return type and parameters go inside angled brackets, with the parameters inside parentheses. If there are no parameters, the parentheses can be left empty.
Updating our earlier example with std::function:

```
#include <functional>
#include <iostream>

int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    std::function<int()> fcnPtr{ &foo }; // declare function pointer that
        returns an int and takes no parameters
    fcnPtr = &goo; // fcnPtr now points to function goo
    std::cout << fcnPtr() << '\n'; // call the function just like normal

    std::function fcnPtr2{ &foo }; // can also use CTAD to infer template
        arguments

    return 0;
}
```

Type inference for function pointers

Much like the auto keyword can be used to infer the type of normal variables, the auto keyword can also infer the type of a function pointer.

```
#include <iostream>

int foo(int x)
{
  return x;
}

int main()
{
  auto fcnPtr{ &foo };
  std::cout << fcnPtr(5) << '\n';

  return 0;
}
```

This works exactly like you'd expect, and the syntax is very clean. The downside is, of course, that all of the details about the function's parameters types and return type are hidden, so it's easier to make a mistake when making a call with the function, or using its return value.

17.2 The stack and the heap

https://www.learncpp.com/cpp-tutorial/the-stack-and-the-heap/
The memory that a program uses is typically divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- The data segment (also called the initialized data segment), where initialized global and static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

17.2.1 The heap segment

In C++, when you use the new operator to allocate memory, this memory is allocated in the application's heap segment.
The address of this memory is passed back by operator new, and can then be stored in a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
int* ptr1 { new int };
int* ptr2 { new int };
// ptr1 and ptr2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is “returned” to the heap and can then be reassigned as future allocation requests are received. Remember that deleting a pointer does not delete the variable, it just returns the memory at the associated address back to the operating system.

The heap has advantages and disadvantages:

- Allocating memory on the heap is comparatively slow.
- Allocated memory stays allocated until it is specifically deallocated (beware memory leaks) or the application ends (at which point the OS should clean it up).
- Dynamically allocated memory must be accessed through a pointer. Dereferencing a pointer is slower than accessing a variable directly.
- Because the heap is a big pool of memory, large arrays, structures, or classes can be allocated here.

17.2.2 The call stack

The **call stack** (usually referred to as "the stack") has a much more interesting role to play. The call stack keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.
The call stack is implemented as a stack data structure. So before we can talk about how the call stack works, we need to understand what a stack data structure is.

17.2.3 The stack data structure

A **data structure** is a programming mechanism for organizing data so that it can be used efficiently. You've already seen several types of data structures, such as arrays and structs. Both of these data structures provide mechanisms for storing data and accessing that data in an efficient way. There are many additional data structures that are commonly used in programming, quite a few of which are implemented in the standard library, and a stack is one of those.
In computer programming, a stack is a container data structure that holds multiple variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish (called random access), a stack is more limited. The operations that can be performed on a stack correspond to the following three things:

- Look at the top item on the stack (usually done via a function called top(), but sometimes called peek())
- Take the top item off of the stack (done via a function called pop())
- Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, the first plate removed from the stack will be the plate you just pushed on last. Last on, first off. As items are pushed onto a stack, the stack grows larger – as items are popped off, the stack grows smaller.

17.2.4 The call stack segment

The call stack segment holds the memory used for the call stack. When the application starts, the main() function is pushed on the call stack by the operating system. Then the program begins executing.
When a function call is encountered, the function is pushed onto the call stack. When the current function ends, that function is popped off the call stack (this process is sometimes called unwinding the stack). Thus, by looking at the functions that are currently on the call stack, we can see all of the functions that were called to get to the current point of execution.
We can make one further optimization: When we pop an item off the call stack, we only have to move the stack pointer down – we don't have to clean up or zero the memory used by the popped stack frame.
This memory is no longer considered to be "on the stack" (the stack pointer will be at or below this address), so it won't be accessed. If we later push a new stack frame to this same memory, it will overwrite the old value we never cleaned up.

The call stack in action

Let's examine in more detail how the call stack works. Here is the sequence of steps that takes place when a function is called:

1. The program encounters a function call.

2. A stack frame is constructed and pushed on the stack. The stack frame consists of:
   - The address of the instruction beyond the function call (called the return address). This is how the CPU remembers where to return to after the called function exits.
   - All function arguments.
   - Memory for any local variables
   - Saved copies of any registers modified by the function that need to be restored when the function returns
3. The CPU jumps to the function's start point.
4. The instructions inside of the function begin executing.

When the function terminates, the following steps happen:

1. Registers are restored from the call stack
2. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
3. The return value is handled.
4. The CPU resumes execution at the return address.

Return values can be handled in a number of different ways, depending on the computer's architecture. Some architectures include the return value as part of the stack frame. Others use CPU registers.
Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off (unwound) when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.
A technical note: on some architectures, the call stack grows away from memory address 0. On others, it grows towards memory address 0. As a consequence, newly pushed stack frames may have a higher or a lower memory address than the previous ones.

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. On Visual Studio for Windows, the default stack size is 1MB. With g++/Clang for Unix variants, it can be as large as 8MB. If the program tries to put too much information on the stack, stack overflow will result. Stack overflow happens when all the memory in the stack has been allocated – in that case, further allocations begin overflowing into other sections of memory.
Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc. . .) On modern operating systems, overflowing the stack will generally cause your OS to issue an access violation and terminate the program.
Here is an example program that will likely cause a stack overflow. You can run it on your system and watch it crash:

```cpp
#include <iostream>

int main()
{
    int stack[10000000];
    std::cout << "hi" << stack[0]; // we'll use stack[0] here so the
        compiler won't optimize the array away

    return 0;
}
```

This program tries to allocate a huge (likely 40MB) array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use.

17.3 Recursive functions

`https://www.learncpp.com/cpp-tutorial/recursion/`
A **recursive function** in C++ is a function that calls itself.

Recursive termination conditions

Recursive function calls generally work just like normal function calls. However, the program above illustrates the most important difference with recursive functions: you must include a recursive termination condition, or they will run “forever” (actually, until the call stack runs out of memory). A **recursive termination** is a condition that, when met, will cause the recursive function to stop calling itself.
Recursive termination generally involves using an if statement. Here is an example:

```cpp
#include <iostream>

void countDown(int count)
{
    std::cout << "push " << count << '\n';

    if (count > 1) // termination condition
        countDown(count-1);

    std::cout << "pop " << count << '\n';
}

int main()
{
    countDown(5);
    return 0;
}
```

Now when we run our program, countDown() will start by outputting the following:

```
push 5
push 4
push 3
push 2
push 1
```

If you were to look at the call stack at this point, you would see the following:

```
countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
main()
```

Thus, this program in total outputs:

```
push 5
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
pop 5
```

It’s worth noting that the “push” outputs happen in forward order since they occur before the recursive function call. The “pop” outputs occur in reverse order because they occur after the recursive function call, as the functions are being popped off the stack (which happens in the reverse order that they were put on).

Recursive algorithms

Recursive programs are often hard to figure out just by looking at them. It’s often instructive to see what happens when we call a recursive function with a particular value.
Recursive functions typically solve a problem by first finding the solution to a subset of the problem (recursively), and then modifying that sub-solution to get to a solution. In the above algorithm, sumTo(value) first solves sumTo(value-1), and then adds the value of variable value to find the solution for sumTo(value).

In many recursive algorithms, some inputs produce trivial outputs. For example, sumTo(1) has the trivial output 1 (you can calculate this in your head), and does not benefit from further recursion. Inputs for which an algorithm trivially produces an output is called a base case. Base cases act as termination conditions for the algorithm. Base cases can often be identified by considering the output for an input of 0, 1, “”, ”, or null.

Recursive vs iterative

One question that is often asked about recursive functions is, “Why use a recursive function if you can do many of the same tasks iteratively (using a for loop or while loop)?”. It turns out that you can always solve a recursive problem iteratively – however, for non-trivial problems, the recursive version is often much simpler to write (and read). For example, while it’s possible to write the Fibonacci function iteratively, it’s a little more difficult! (Try it!)
Iterative functions (those using a for-loop or while-loop) are almost always more efficient than their recursive counterparts. This is because every time you call a function there is some amount of overhead that takes place in pushing and popping stack frames. Iterative functions avoid this overhead.
That’s not to say iterative functions are always a better choice. Sometimes the recursive implementation of a function is so much cleaner and easier to follow that incurring a little extra overhead is more than worth it for the benefit in maintainability, particularly if the algorithm doesn’t need to recurse too many times to find a solution.
In general, recursion is a good choice when most of the following are true:

- The recursive code is much simpler to implement.
- The recursion depth can be limited (e.g. there’s no way to provide an input that will cause it to recurse down 100,000 levels).
- The iterative version of the algorithm requires managing a stack of data.
- This isn’t a performance-critical section of code.

Best practice: Generally favor iteration over recursion, except when recursion really makes sense.

17.4 Command line arguments

https://www.learncpp.com/cpp-tutorial/command-line-arguments/

17.5 Lambda functions

https://www.learncpp.com/cpp-tutorial/introduction-to-lambdas-anonymous-functions/
A **lambda expression** (also called a lambda or closure) allows us to define an anonymous function inside another function. The nesting is important, as it allows us both to avoid namespace naming pollution, and to define the function as close to where it is used as possible (providing additional context).

17.5.1 Definition

The syntax for lambdas is one of the weirder things in C++, and takes a bit of getting used to. Lambdas take the form:

```
[ captureClause ] ( parameters ) -> returnType
{
    statements;
}
```

- The capture clause can be empty if no captures are needed.
- The parameter list can be empty if no parameters are required. It can also be omitted entirely unless a return type is specified.
- The return type is optional, and if omitted, auto will be assumed (thus using type deduction used to determine the return type). While we previously noted that type deduction for function return types should be avoided, in this context, it’s fine to use (because these functions are typically so trivial).

Also note that lambdas (being anonymous) have no name, so we don't need to provide one.
Consider this example:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  constexpr std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // Define the function right where we use it.
  auto found{ std::find_if(arr.begin(), arr.end(),
                           [](std::string_view str) // here's our lambda, no capture clause
                           {
                             return str.find("nut") != std::string_view::npos;
                           }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

This works just like in the case where the function is defined as a function pointer, and produces an identical result while being much cleaner.

17.5.2 Type of a lambda

Writing a lambda in the same line as it's used can sometimes make code harder to read. Much like we can initialize a variable with a literal value (or a function pointer) for use later, we can also initialize a lambda variable with a lambda definition and then use it later. A named lambda along with a good function name can make code easier to read.
For example, in the following snippet, we're using std::all_of to check if all elements of an array are even:

```cpp
// Bad: We have to read the lambda to understand what's happening.
return std::all_of(array.begin(), array.end(), [](int i){ return ((i % 2) == 0); });
```

We can improve the readability of this as follows:

```cpp
// Good: Instead, we can store the lambda in a named variable and pass it to the function.
auto isEven{
  [](int i)
  {
    return (i % 2) == 0;
  }
};

return std::all_of(array.begin(), array.end(), isEven);
```

But what is the type of lambda `isEven`?
As it turns out, lambdas don't have a type that we can explicitly use. When we write a lambda, the compiler generates a unique type just for the lambda that is not exposed to us.
Although we don't know the type of a lambda, there are several ways of storing a lambda for use post-definition. If the lambda has an empty capture clause (nothing between the hard brackets []), we can use

a regular function pointer. std::function or type deduction via the auto keyword will also work (even if the lambda has a non-empty capture clause).

```cpp
#include <functional>

int main()
{
  // A regular function pointer. Only works with an empty capture clause
      (empty []).
  double (*addNumbers1)(double, double){
    [](double a, double b) {
      return a + b;
    }
  };

  addNumbers1(1, 2);

  // Using std::function. The lambda could have a non-empty capture clause
      (discussed next lesson).
  std::function addNumbers2{ // note: pre-C++17, use
      std::function<double(double, double)> instead
    [](double a, double b) {
      return a + b;
    }
  };

  addNumbers2(3, 4);

  // Using auto. Stores the lambda with its real type.
  auto addNumbers3{
    [](double a, double b) {
      return a + b;
    }
  };

  addNumbers3(5, 6);

  return 0;
}
```

17.5.3 Passing Lambdas

If we want to pass a lambda to a function there are 4 options:

```cpp
#include <functional>
#include <iostream>

// Case 1: use a `std::function` parameter
void repeat1(int repetitions, const std::function<void(int)>& fn)
{
    for (int i{ 0 }; i < repetitions; ++i)
        fn(i);
}

// Case 2: use a function template with a type template parameter
template <typename T>
void repeat2(int repetitions, const T& fn)
{
    for (int i{ 0 }; i < repetitions; ++i)
        fn(i);
}

// Case 3: use the abbreviated function template syntax (C++20)
void repeat3(int repetitions, const auto& fn)
{
    for (int i{ 0 }; i < repetitions; ++i)
```

```
        fn(i);
}

// Case 4: use function pointer (only for lambda with no captures)
void repeat4(int repetitions, void (*fn)(int))
{
    for (int i{ 0 }; i < repetitions; ++i)
        fn(i);
}

int main()
{
    auto lambda = [](int i)
    {
        std::cout << i << '\n';
    };

    repeat1(3, lambda);
    repeat2(3, lambda);
    repeat3(3, lambda);
    repeat4(3, lambda);

    return 0;
}
```

In case 1, our function parameter is a std::function. This is nice because we can explicitly see what the parameters and return type of the std::function are. However, this requires the lambda to be implicitly converted whenever the function is called, which adds some overhead. This method also has the benefit of being separable into a declaration (in a header) and a definition (in a .cpp file) if that's desirable.
In case 2, we're using a function template with type template parameter T. When the function is called, a function will be instantiated where T matches the actual type of the lambda. This is more efficient, but the parameters and return type of T are not obvious.
In case 3, we use C++20's auto to invoke the abbreviated function template syntax. This generates a function template identical to case 2.
In case 4, the function parameter is a function pointer. Since a lambda with no captures will implicitly convert to a function pointer, we can pass a lambda with no captures to this function.

Best practice: When storing a lambda in a variable, use auto as the variable's type.

When passing a lambda to a function:

If C++20 capable, use auto as the parameter's type. Otherwise, use a function with a type template parameter or std::function parameter (or a function pointer if the lambda has no captures).

17.5.4 Constexpr lambdas

As of C++17, lambdas are implicitly constexpr if the result satisfies the requirements of a constant expression. This generally requires two things:

- The lambda must either have no captures, or all captures must be constexpr.
- The functions called by the lambda must be constexpr. Note that many standard library algorithms and math functions weren't made constexpr until C++20 or C++23.

Standard library function objects

For common operations (e.g. addition, negation, or comparison) you don't need to write your own lambdas, because the standard library comes with many basic callable objects that can be used instead. These are defined in the ¡functional¿ header.

```
#include <algorithm>
#include <array>
#include <iostream>
#include <functional> // for std::greater

```

```cpp
int main()
{
  std::array arr{ 13, 90, 99, 5, 40, 80 };

  // Pass std::greater to std::sort
  std::sort(arr.begin(), arr.end(), std::greater{}); // note: need curly
      braces to instantiate object

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  ret
```

Lambdas and the algorithm library may seem unnecessarily complicated when compared to a solution that uses a loop. However, this combination can allow some very powerful operations in just a few lines of code, and can be more readable than writing your own loops. On top of that, the algorithm library features powerful and easy-to-use parallelism, which you won't get with loops. Upgrading source code that uses library functions is easier than upgrading code that uses loops.
Lambdas are great, but they don't replace regular functions for all cases. Prefer regular functions for non-trivial and reusable cases.

17.6 Lambda Captures

https://www.learncpp.com/cpp-tutorial/lambda-captures/
Now, let's modify the nut example and let the user pick a substring to search for. This isn't as intuitive as you might expect.

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut",
      "lemon" };

  // Ask the user what to search for.
  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  auto found{ std::find_if(arr.begin(), arr.end(), [](std::string_view
      str) {
    // Search for @search rather than "nut".
    return str.find(search) != std::string_view::npos; // Error: search
        not accessible in this scope
  }) };

  if (found == arr.end())
  {
    std::cout << "Not found\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

This code won't compile. Unlike nested blocks, where any identifier accessible in the outer block is accessible in the nested block, lambdas can only access certain kinds of objects that have been defined outside the lambda. This includes:
Objects with static (or thread local) storage duration (this includes global variables and static locals) Objects that are constexpr (explicitly or implicitly)

17.6.1 The capture clause

The **capture clause** is used to (indirectly) give a lambda access to variables available in the surrounding scope that it normally would not have access to. All we need to do is list the entities we want to access from within the lambda as part of the capture clause. We can modify
When a lambda definition is executed, for each variable that the lambda captures, a clone of that variable is made (with an identical name) inside the lambda. These cloned variables are initialized from the outer scope variables of the same name at this point. In this case, we want to give our lambda access to the value of variable search, so we add it to the capture clause:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut",
      "lemon" };

  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  // Capture @search                                vvvvvv
  auto found{ std::find_if(arr.begin(), arr.end(),
      [search](std::string_view str) {
    return str.find(search) != std::string_view::npos;
  }) };

  if (found == arr.end())
  {
    std::cout << "Not found\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  ret
```

Captures are treated as const by default

When a lambda is called, operator() is invoked. By default, this operator() treats captures as const, meaning the lambda is not allowed to modify those captures.

17.6.2 Mutable captures

To allow modifications of variables that were captured, we can mark the lambda as mutable:

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    [ammo]() mutable { // now mutable
      // We're allowed to modify ammo now
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();
  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

17.6.3 Capture by reference

Much like functions can change the value of arguments passed by reference, we can also capture variables by reference to allow our lambda to affect the value of the argument.
To capture a variable by reference, we prepend an ampersand (&) to the variable name in the capture. Unlike variables that are captured by value, variables that are captured by reference are non-const, unless the variable they're capturing is const. Capture by reference should be preferred over capture by value whenever you would normally prefer passing an argument to a function by reference (e.g. for non-fundamental types).
Here's the above code with ammo captured by reference:

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    // We don't need mutable anymore
    [&ammo]() { // &ammo means ammo is captured by reference
      // Changes to ammo will affect main's ammo
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

17.6.4 Capturing multiple variables

Multiple variables can be captured by separating them with a comma. This can include a mix of variables captured by value or by reference:

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Capture health and armor by value, and enemies by reference.
[health, armor, &enemies](){};
```

17.6.5 Default captures

Having to explicitly list the variables you want to capture can be burdensome. If you modify your lambda, you may forget to add or remove captured variables. Fortunately, we can enlist the compiler's help to auto-generate a list of variables we need to capture.
A default capture (also called a capture-default) captures all variables that are mentioned in the lambda. Variables not mentioned in the lambda are not captured if a default capture is used.

- To capture all used variables by value, use a capture value of =.
- To capture all used variables by reference, use a capture value of &.

Here's an example of using a default capture by value:

```cpp
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  auto found{ std::find_if(areas.begin(), areas.end(),
                           [=](int knownArea) { // will default capture
                               width and height by value
                             return width * height == knownArea; //
                               because they're mentioned here
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

Default captures can be mixed with normal captures. We can capture some variables by value and others by reference, but each variable can only be captured once.

Defining new variables in the lambda-capture

Sometimes we want to capture a variable with a slight modification or declare a new variable that is only visible in the scope of the lambda. We can do so by defining a variable in the lambda-capture without specifying its type.

```cpp
#include <array>
#include <iostream>
#include <algorithm>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  // We store areas, but the user entered width and height.
  // We need to calculate the area before we can search for it.
  auto found{ std::find_if(areas.begin(), areas.end(),
                           // Declare a new variable that's visible only
                               to the lambda.
                           // The type of userArea is automatically
                               deduced to int.
                           [userArea{ width * height }](int knownArea) {
                             return userArea == knownArea;
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

Best practice: Only initialize variables in the capture if their value is short and their type is obvious. Otherwise it's best to define the variable outside of the lambda and capture it.

Dangling captured variables

Variables are captured at the point where the lambda is defined. If a variable captured by reference dies before the lambda, the lambda will be left holding a dangling reference.

Chapter 18

Algorithms

18.1 Sorting an array using selection sort

Sorting an array is the process of arranging all of the elements in the array in a particular order. There are many different cases in which sorting an array can be useful. For example, your email program generally displays emails in order of time received, because more recent emails are typically considered more relevant. When you go to your contact list, the names are typically in alphabetical order, because it's easier to find the name you are looking for that way. Both of these presentations involve sorting data before presentation.
Sorting an array can make searching an array more efficient, not only for humans, but also for computers.
Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some predefined criteria. The order in which these elements are compared differs depending on which sorting algorithm is used. The criteria depends on how the list will be sorted (e.g. in ascending or descending order).
To swap two elements, we can use the std::swap() function from the C++ standard library, which is defined in the utility header.

```cpp
#include <iostream>
#include <utility>

int main()
{
    int x{ 2 };
    int y{ 4 };
    std::cout << "Before swap: x = " << x << ", y = " << y << '\n';
    std::swap(x, y); // swap the values of x and y
    std::cout << "After swap:  x = " << x << ", y = " << y << '\n';

    return 0;
}
```

18.1.1 Selection sort

There are many ways to sort an array. Selection sort is probably the easiest sort to understand, which makes it a good candidate for teaching even though it is one of the slowest sorts.
[...]
Selection sort in C++

```cpp
#include <iostream>
#include <iterator>
#include <utility>

int main()
{
  int array[]{ 30, 50, 20, 10, 40 };
  constexpr int length{ static_cast<int>(std::size(array)) };

  // Step through each element of the array
  // (except the last one, which will already be sorted by the time we get
      there)
  for (int startIndex{ 0 }; startIndex < length - 1; ++startIndex)
  {
    // smallestIndex is the index of the smallest element we've encountered this iteration
    // Start by assuming the smallest element is the first element of this iteration
    int smallestIndex{ startIndex };

    // Then look for a smaller element in the rest of the array
    for (int currentIndex{ startIndex + 1 }; currentIndex < length; ++currentIndex)
    {
      // If we've found an element that is smaller than our previously found smallest
      if (array[currentIndex] < array[smallestIndex])
        // then keep track of it
        smallestIndex = currentIndex;
    }

    // smallestIndex is now the index of the smallest element in the remaining array
                // swap our start element with our smallest element (this sorts it into the correct place)
    std::swap(array[startIndex], array[smallestIndex]);
  }

  // Now that the whole array is sorted, print our sorted array as proof it works
  for (int index{ 0 }; index < length; ++index)
    std::cout << array[index] << ' ';

  std::cout << '\n';

  return 0;
}
```

The most confusing part of this algorithm is the loop inside of another loop (called a nested loop). The outside loop (startIndex) iterates through each element one by one. For each iteration of the outer loop, the inner loop (currentIndex) is used to find the smallest element in the remaining array (starting from startIndex+1). smallestIndex keeps track of the index of the smallest element found by the inner loop. Then smallestIndex is swapped with startIndex. Finally, the outer loop (startIndex) advances one element, and the process is repeated.
Hint: If you're having trouble figuring out how the above program works, it can be helpful to work through a sample case on a piece of paper. Write the starting (unsorted) array elements horizontally at the top of the paper. Draw arrows indicating which elements startIndex, currentIndex, and smallestIndex are indexing. Manually trace through the program and redraw the arrows as the indices change. For each iteration of the outer loop, start a new line showing the current state of the array.

18.1.2 std::sort

Because sorting arrays is so common, the C++ standard library includes a sorting function named std::sort. std::sort lives in the ¡algorithm¿ header, and can be invoked on an array like so:

```
#include <algorithm> // for std::sort
#include <iostream>
#include <iterator> // for std::size

int main()
{
  int array[]{ 30, 50, 20, 10, 40 };

  std::sort(std::begin(array), std::end(array));

  for (int i{ 0 }; i < static_cast<int>(std::size(array)); ++i)
    std::cout << array[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

Iterating through an array (or other structure) of data is quite a common thing to do in programming. And so far, we've covered many different ways to do so: with loops and an index (for-loops and while loops), with pointers and pointer arithmetic, and with range-based for-loops:

18.2 Iterators

An iterator is an object designed to traverse through a container (e.g. the values in an array, or the characters in a string), providing access to each element along the way.
A container may provide different kinds of iterators. For example, an array container might offer a forwards iterator that walks through the array in forward order, and a reverse iterator that walks through the array in reverse order.
Once the appropriate type of iterator is created, the programmer can then use the interface provided by the iterator to traverse and access elements without having to worry about what kind of traversal is being done or how the data is being stored in the container. And because C++ iterators typically use the same interface for traversal (operator++ to move to the next element) and access (operator* to access the current element), we can iterate through a wide variety of different container types using a consistent method.

Pointers as an iterator

The simplest kind of iterator is a pointer, which (using pointer arithmetic) works for data stored sequentially in memory. Let's revisit a simple array traversal using a pointer and pointer arithmetic:

```
#include <array>
#include <iostream>

int main()
{
    std::array arr{ 0, 1, 2, 3, 4, 5, 6 };

    auto begin{ &arr[0] };
    // note that this points to one spot beyond the last element
    auto end{ begin + std::size(arr) };

    // for-loop with pointer
    for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next
        element
    {
        std::cout << *ptr << ' '; // Indirection to get value of current
            element
    }
    std::cout << '\n';

    return 0;
}
```

18.2.1 Standard library iterators

Iterating is such a common operation that all standard library containers offer direct support for iteration. Instead of calculating our own begin and end points, we can simply ask the container for the begin and end points via member functions conveniently named begin() and end():

```
#include <array>
#include <iostream>

int main()
{
    std::array array{ 1, 2, 3 };

```

```cpp
    // Ask our array for the begin and end points (via the begin and end member functions).
    auto begin{ array.begin() };
    auto end{ array.end() };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
    {
        std::cout << *p << ' '; // Indirection to get value of current element.
    }
    std::cout << '\n';

    return 0;
}
```

The iterator header also contains two generic functions (std::begin and std::end) that can be used.

```cpp
#include <array>    // includes <iterator>
#include <iostream>

int main()
{
    std::array array{ 1, 2, 3 };

    // Use std::begin and std::end to get the begin and end points.
    auto begin{ std::begin(array) };
    auto end{ std::end(array) };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element
    {
        std::cout << *p << ' '; // Indirection to get value of current element
    }
    std::cout << '\n';

    return 0;
}
```

operator<vs operator!= for iterators

With iterators, it is conventional to use operator!= to test whether the iterator has reached the end element:

```cpp
for (auto p{ begin }; p != end; ++p)
```

This is because some iterator types are not relationally comparable. operator!= works with all iterator types.

18.2.2 range-based for loops

```cpp
#include <array>
#include <iostream>

int main()
{
    std::array array{ 1, 2, 3 };

    // This does exactly the same as the loop we used before.
    for (int i : array)
    {
        std::cout << i << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

Behind the scenes, the range-based for-loop calls begin() and end() of the type to iterate over. std::array has begin and end member functions, so we can use it in a range-based loop. C-style fixed arrays can be used with std::begin and std::end functions, so we can loop through them with a range-based loop as well. Dynamic C-style arrays (or decayed C-style arrays) don't work though, because there is no std::end function for them (because the type information doesn't contain the array's length).

18.2.3 Iterator invalidation (dangling iterators)

Much like pointers and references, iterators can be left "dangling" if the elements being iterated over change address or are destroyed. When this happens, we say the iterator has been invalidated. Accessing an invalidated iterator produces undefined behavior.
Since range-based for-loops use iterators behind the scenes, we must be careful not to do anything that invalidates the iterators of the container we are actively traversing:

```cpp
#include <vector>

int main()
{
    std::vector v { 0, 1, 2, 3 };

    for (auto num : v) // implicitly iterates over v
    {
        if (num % 2 == 0)
            v.push_back(num + 1); // when this invalidates the iterators
                of v, undefined behavior will result
    }

    return 0;
}
```

Here's another example of iterator invalidation:

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector v{ 1, 2, 3, 4, 5, 6, 7 };

  auto it{ v.begin() };

  ++it; // move to second element
  std::cout << *it << '\n'; // ok: prints 2

  v.erase(it); // erase the element currently being iterated over

  // erase() invalidates iterators to the erased element (and subsequent
     elements)
  // so iterator "it" is now invalidated

  ++it; // undefined behavior
  std::cout << *it << '\n'; // undefined behavior

  return 0;
}
```

18.3 Standard library algorithms

New programmers typically spend a lot of time writing custom loops to perform relatively simple tasks, such as sorting or counting or searching arrays. These loops can be problematic, both in terms of how easy it is to make an error, and in terms of overall maintainability, as loops can be hard to understand.
Because searching, counting, and sorting are such common operations to do, the C++ standard library comes with a bunch of functions to do these things in just a few lines of code. Additionally, these standard library

functions come pre-tested, are efficient, work on a variety of different container types, and many support parallelization (the ability to devote multiple CPU threads to the same task in order to complete it faster).
The functionality provided in the algorithms library generally fall into one of three categories:

- **Inspectors** – Used to view (but not modify) data in a container. Examples include searching and counting.
- **Mutators** – Used to modify data in a container. Examples include sorting and shuffling.
- **Facilitators** – Used to generate a result based on values of the data members. Examples include objects that multiply values, or objects that determine what order pairs of elements should be sorted in.

These algorithms live in the algorithms library. In this lesson, we'll explore some of the more common algorithms – but there are many more, and we encourage you to read through the linked reference to see everything that's available!

18.3.1 `std::find`

std::find searches for the first occurrence of a value in a container. std::find takes 3 parameters: an iterator to the starting element in the sequence, an iterator to the ending element in the sequence, and a value to search for. It returns an iterator pointing to the element (if it is found) or the end of the container (if the element is not found).
For example:

```cpp
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    std::cout << "Enter a value to search for and replace with: ";
    int search{};
    int replace{};
    std::cin >> search >> replace;

    // Input validation omitted

    // std::find returns an iterator pointing to the found element (or the
        end of the container)
    // we'll store it in a variable, using type inference to deduce the
        type of
    // the iterator (since we don't care)
    auto found{ std::find(arr.begin(), arr.end(), search) };

    // Algorithms that don't find what they were looking for return the
        end iterator.
    // We can access it by using the end() member function.
    if (found == arr.end())
    {
        std::cout << "Could not find " << search << '\n';
    }
    else
    {
        // Override the found element.
        *found = replace;
    }

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

18.3.2 std::count

std::count and std::count_if search for all occurrences of an element or an element fulfilling a condition. In the following example, we'll count how many elements contain the substring "nut":

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

bool containsNut(std::string_view str)
{
  return str.find("nut") != std::string_view::npos;
}

int main()
{
  std::array<std::string_view, 5> arr{ "apple", "banana", "walnut",
      "lemon", "peanut" };

  auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };

  std::cout << "Counted " << nuts << " nut(s)\n";

  return 0;
}
```

Output

```
Counted 2 nut(s)
```

18.3.3 std::sort

Let's use std::sort to sort an array in reverse order using a custom comparison function named greater:

```cpp
#include <algorithm>
#include <array>
#include <iostream>

bool greater(int a, int b)
{
    // Order @a before @b if @a is greater than @b.
    return (a > b);
}

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    // Pass greater to std::sort
    std::sort(arr.begin(), arr.end(), greater);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Once again, instead of writing our own custom loop functions, we can sort our array however we like in just a few lines of code!
Our greater function needs 2 arguments, but we're not passing it any, so where do they come from? When we use a function without parentheses (), it's only a function pointer, not a call. You might remember this from when we tried to print a function without parentheses and std::cout printed "1". std::sort uses this pointer and calls the actual greater function with any 2 elements of the array. We don't know which elements greater will be called with, because it's not defined which sorting algorithm std::sort is using under the hood. We talk more about function pointers in a later chapter.

18.3.4 std::for_each

std::for_each takes a list as input and applies a custom function to every element. This is useful when we want to perform the same operation to every element in a list.

```cpp
#include <algorithm>
#include <array>
#include <iostream>

void doubleNumber(int& i)
{
    i *= 2;
}

int main()
{
    std::array arr{ 1, 2, 3, 4 };

    std::for_each(arr.begin(), arr.end(), doubleNumber);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

This often seems like the most unnecessary algorithm to new developers, because equivalent code with a range-based for-loop is shorter and easier. But there are benefits to std::for_each. Let's compare std::for_each to a range-based for-loop.

```cpp
std::ranges::for_each(arr, doubleNumber); // Since C++20, we don't have to
    use begin() and end().
// std::for_each(arr.begin(), arr.end(), doubleNumber); // Before C++20

for (auto& i : arr)
{
    doubleNumber(i);
}
```

With std::for_each, our intentions are clear. Call doubleNumber with each element of arr. In the range-based for-loop, we have to add a new variable, i. This leads to several mistakes that a programmer could do when they're tired or not paying attention. For one, there could be an implicit conversion if we don't use auto. We could forget the ampersand, and doubleNumber wouldn't affect the array. We could accidentally pass a variable other than i to doubleNumber. These mistakes cannot happen with std::for_each.
Additionally, std::for_each can skip elements at the beginning or end of a container, for example to skip the first element of arr, std::next can be used to advance begin to the next element.

```cpp
std::for_each(std::next(arr.begin()), arr.end(), doubleNumber);
// Now arr is [1, 4, 6, 8]. The first element wasn't doubled.
```

This isn't possible with a range-based for-loop.
Like many algorithms, std::for_each can be parallelized to achieve faster processing, making it better suited for large projects and big data than a range-based for-loop.

18.4 Performance and order of execution

Many of the algorithms in the algorithms library make some kind of guarantee about how they will execute. Typically these are either performance guarantees, or guarantees about the order in which they will execute. For example, std::for_each guarantees that each element will only be accessed once, and that the elements will be accessed in forwards sequential order.
While most algorithms provide some kind of performance guarantee, fewer have order of execution guarantees. For such algorithms, we need to be careful not to make assumptions about the order in which elements will be accessed or processed.
For example, if we were using a standard library algorithm to multiply the first value by 1, the second value by 2, the third by 3, etc... we'd want to avoid using any algorithms that didn't guarantee a forwards sequential execution order!
The following algorithms guarantee sequential execution: std::for_each, std::copy, std::copy_backward, std::move, and std::move_backward. Many other algorithms (particular those that use a forward iterator) are implicitly sequential due to the forward iterator requirement.

Best practice: Favor using functions from the algorithms library over writing your own functionality to do the same thing.

Best practice: Before using a particular algorithm, make sure performance and execution order guarantees work for your particular use case.

18.5 Timing your code

Chapter 19

Dinamic allocation

19.1 Dynamic memory allocation

19.1.1 The need for dynamic memory allocation

C++ supports three basic types of memory allocation, of which you've already seen two.

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
- **Dynamic memory allocation** is the topic of this article.

Both static and automatic allocation have two things in common:

- The size of the variable / array must be known at compile time.
- Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

Most of the time, this is just fine. However, you will come across situations where one or both of these constraints cause problems, usually when dealing with external (user or file) input.
For example, we may want to use a string to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.
If we have to declare the size of everything at compile time, the best we can do is try to make a guess the maximum size of variables we'll need and hope that's enough.
This is a poor solution for at least four reasons:
First, it leads to wasted memory if the variables aren't actually used. For example, if we allocate 25 chars for every name, but names on average are only 12 chars long, we're using over twice what we really need. Or consider the rendering array above: if a rendering only uses 10,000 polygons, we have 20,000 Polygons worth of memory not being used!
Second, how do we tell which bits of memory are actually used? For strings, it's easy: a string that starts with a 0 is clearly not being used. But what about monster[24]? Is it alive or dead right now? Has it even been initialized in the first place? That necessitates having some way to tell the status of each monster, which adds complexity and can use up additional memory.
Third, most normal variables (including fixed arrays) are allocated in a portion of memory called the stack. The amount of stack memory for a program is generally quite small – Visual Studio defaults the stack size to 1MB. If you exceed this number, stack overflow will result, and the operating system will probably close down the program.
Fourth, and most importantly, it can lead to artificial limitations and/or array overflows. What happens when the user tries to read in 600 records from disk, but we've only allocated memory for a maximum of 500 records? Either we have to give the user an error, only read the 500 records, or (in the worst case where we don't handle this case at all) overflow the record array and watch something bad happen.

Fortunately, these problems are easily addressed via dynamic memory allocation. **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory – instead, it is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

19.1.2 Dynamically allocating single variables

To allocate a single variable dynamically, we use the scalar (non-array) form of the **new** operator:

```
new int; // dynamically allocate an integer (and discard the result)
```

In the above case, we're requesting an integer's worth of memory from the operating system. The new operator creates the object using that memory, and then returns a pointer containing the address of the memory that has been allocated.
Most often, we'll assign the return value to our own pointer variable so we can access the allocated memory later.

```
int* ptr{ new int }; // dynamically allocate an integer and assign the
    address to ptr so we can access it later
```

Note that accessing heap-allocated objects is generally slower than accessing stack-allocated objects. Because the compiler knows the address of stack-allocated objects, it can go directly to that address to get a value. Heap allocated objects are typically accessed via pointer. This requires two steps: one to get the address of the object (from the pointer), and another to get the value.

How does dynamic memory allocation work?

Your computer has memory (probably lots of it) that is available for applications to use. When you run an application, your operating system loads the application into some of that memory. This memory used by your application is divided into different areas, each of which serves a different purpose. One area contains your code. Another area is used for normal operations (keeping track of which functions were called, creating and destroying global and local variables, etc...). We'll talk more about those later. However, much of the memory available just sits there, waiting to be handed out to programs that request it.
When you dynamically allocate memory, you're asking the operating system to reserve some of that memory for your program's use. If it can fulfill this request, it will return the address of that memory to your application. From that point forward, your application can use this memory as it wishes. When your application is done with the memory, it can return the memory back to the operating system to be given to another program.
Unlike static or automatic memory, the program itself is responsible for requesting and disposing of dynamically allocated memory.

Initializing a dynamically allocated variable

When you dynamically allocate a variable, you can also initialize it via direct initialization or uniform initialization:

```
int* ptr1{ new int (5) }; // use direct initialization
int* ptr2{ new int { 6 } }; // use uniform initialization
```

Deleting a single variable

When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. For single variables, this is done via the scalar (non-array) form of the **delete** operator:

```
// assume ptr has previously been allocated with operator new
delete ptr; // return the memory pointed to by ptr to the operating system
ptr = nullptr; // set ptr to be a null pointer
```

What does it mean to delete memory?

The delete operator does not actually delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later).

Although the syntax makes it look like we're deleting a variable, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value (e.g. nullptr) just like any other variable.
Note that deleting a pointer that is not pointing to dynamically allocated memory may cause bad things to happen.

19.1.3 Dangling pointers

C++ does not make any guarantees about what will happen to the contents of deallocated memory, or to the value of the pointer being deleted. In most cases, the memory returned to the operating system will contain the same values it had before it was returned, and the pointer will be left pointing to the now deallocated memory.
A pointer that is pointing to deallocated memory is called a **dangling pointer**. Dereferencing or deleting a dangling pointer will lead to undefined behavior. Consider the following program:

```cpp
#include <iostream>

int main()
{
    int* ptr{ new int }; // dynamically allocate an integer
    *ptr = 7; // put a value in that memory location

    delete ptr; // return the memory to the operating system.  ptr is now
        a dangling pointer.

    std::cout << *ptr; // Dereferencing a dangling pointer will cause
        undefined behavior
    delete ptr; // trying to deallocate the memory again will also lead to
        undefined behavior.

    return 0;
}
```

Deallocating memory may create multiple dangling pointers. Consider the following example:

```cpp
#include <iostream>

int main()
{
    int* ptr{ new int{} }; // dynamically allocate an integer
    int* otherPtr{ ptr }; // otherPtr is now pointed at that same memory
        location

    delete ptr; // return the memory to the operating system.  ptr and
        otherPtr are now dangling pointers.
    ptr = nullptr; // ptr is now a nullptr

    // however, otherPtr is still a dangling pointer!

    return 0;
}
```

There are a few best practices that can help here.
First, try to avoid having multiple pointers point at the same piece of dynamic memory. If this is not possible, be clear about which pointer “owns” the memory (and is responsible for deleting it) and which pointers are just accessing it.
Second, when you delete a pointer, if that pointer is not going out of scope immediately afterward, set the pointer to nullptr. We'll talk more about null pointers, and why they are useful in a bit.

Best practice: Set deleted pointers to nullptr unless they are going out of scope immediately afterward.

```cpp
  int *ptr{new int{0}};
  delete ptr;
  ptr = nullptr;
```

Operator new can fail

When requesting memory from the operating system, in rare circumstances, the operating system may not have any memory to grant the request with.
By default, if new fails, a bad_alloc exception is thrown. If this exception isn't properly handled (and it won't be, since we haven't covered exceptions or exception handling yet), the program will simply terminate (crash) with an unhandled exception error.
In many cases, having new throw an exception (or having your program crash) is undesirable, so there's an alternate form of new that can be used instead to tell new to return a null pointer if memory can't be allocated.
This is done by adding the constant std::nothrow between the new keyword and the allocation type:

```
int* value { new (std::nothrow) int }; // value will be set to a null
    pointer if the integer allocation fails
```

In the above example, if new fails to allocate memory, it will return a null pointer instead of the address of the allocated memory.
Because asking new for memory only fails rarely (and almost never in a dev environment), it's common to forget to do this check!

Null pointers and dynamic memory allocation

Null pointers (pointers set to nullptr) are particularly useful when dealing with dynamic memory allocation.
In the context of dynamic memory allocation, a null pointer basically says "no memory has been allocated to this pointer". This allows us to do things like conditionally allocate memory:

```
// If ptr isn't already allocated, allocate it
if (!ptr)
    ptr = new int;
```

Deleting a null pointer has no effect. Thus, you can just write:

```
delete ptr;
```

If ptr is non-null, the dynamically allocated memory will be deleted. If ptr is null, nothing will happen.

19.1.4 Memory leaks

Dynamically allocated memory stays allocated until it is explicitly deallocated or until the program ends (and the operating system cleans it up, assuming your operating system does that). However, the pointers used to hold dynamically allocated memory addresses follow the normal scoping rules for local variables. This mismatch can create interesting problems.
Consider the following function:

```
void doSomething()
{
    int* ptr{ new int{} };
}
```

This function allocates an integer dynamically, but never frees it using delete. Because pointers variables are just normal variables, when the function ends, ptr will go out of scope. And because ptr is the only variable holding the address of the dynamically allocated integer, when ptr is destroyed there are no more references to the dynamically allocated memory. This means the program has now "lost" the address of the dynamically allocated memory. As a result, this dynamically allocated integer can not be deleted.
This is called a **memory leak**. Memory leaks happen when your program loses the address of some bit of dynamically allocated memory before giving it back to the operating system. When this happens, your program can't delete the dynamically allocated memory, because it no longer knows where it is. The operating system also can't use this memory, because that memory is considered to be still in use by your program.
Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the operating system able to clean up and "reclaim" all leaked memory.
Although memory leaks can result from a pointer going out of scope, there are other ways that memory leaks can result. For example, a memory leak can occur if a pointer holding the address of the dynamically allocated memory is assigned another value:

```cpp
int value = 5;
int* ptr{ new int{} }; // allocate memory
ptr = &value; // old address lost, memory leak results
```

Relatedly, it is also possible to get a memory leak via double-allocation:

```cpp
int* ptr{ new int{} };
ptr = new int{}; // old address lost, memory leak results
```

This can be fixed by deleting the pointer before reassigning it:

```cpp
int value{ 5 };
int* ptr{ new int{} }; // allocate memory
delete ptr; // return memory back to operating system
ptr = &value; // reassign pointer to address of value
```

19.2 Dynamically allocating C-style arrays

To dynamically allocate arrays the sintax is slightly different, but we will not be covering it since we already discouraged the use of C-style arrays.
While you can dynamically allocate a std::array, you're usually better off using a non-dynamically allocated std::vector in this case.

19.3 Destructors

A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. Whereas constructors are designed to initialize a class, destructors are designed to help clean up. When an object goes out of scope normally, or a dynamically allocated object is explicitly deleted using the delete keyword, the class destructor is automatically called (if it exists) to do any necessary clean up before the object is removed from memory. For simple classes (those that just initialize the values of normal member variables), a destructor is not needed because C++ will automatically clean up the memory for you.
However, if your class object is holding any resources (e.g. dynamic memory, or a file or database handle), or if you need to do any kind of maintenance before the object is destroyed, the destructor is the perfect place to do so, as it is typically the last thing to happen before the object is destroyed.

19.3.1 Destructor naming

Like constructors, destructors have specific naming rules:
The destructor must have the same name as the class, preceded by a tilde ( ). The destructor can not take arguments. The destructor has no return type. A class can only have a single destructor.
Generally you should not call a destructor explicitly (as it will be called automatically when the object is destroyed), since there are rarely cases where you'd want to clean up an object more than once. However, destructors may safely call other member functions since the object isn't destroyed until after the destructor executes.
For eexample:

```cpp
#include <iostream>
#include <cassert>
#include <cstddef>

class IntArray
{
private:
  int* m_array{};
  int m_length{};

public:
  IntArray(int length) // constructor
  {
    assert(length > 0);

    m_array = new int[static_cast<std::size_t>(length)]{};
    m_length = length;
  }

  ~IntArray() // destructor
  {
    // Dynamically delete the array we allocated earlier
    delete[] m_array;
  }

  void setValue(int index, int value) { m_array[index] = value; }
  int getValue(int index) { return m_array[index]; }

  int getLength() { return m_length; }
};

int main()
{
  IntArray ar ( 10 ); // allocate 10 integers
  for (int count{ 0 }; count < ar.getLength(); ++count)
    ar.setValue(count, count+1);

  std::cout << "The value of element 5 is: " << ar.getValue(5) << '\n';

  return 0;
} // ar is destroyed here, so the ~IntArray() destructor function is
   called here
```

This program produces the result:

```
The value of element 5 is: 6
```

Constructor and destructor timing

As mentioned previously, the constructor is called when an object is created, and the destructor is called when an object is destroyed. In the following example, we use cout statements inside the constructor and destructor to show this:

```
#include <iostream>

class Simple
{
private:
    int m_nID{};

public:
    Simple(int nID)
        : m_nID{ nID }
    {
        std::cout << "Constructing Simple " << nID << '\n';
    }

    ~Simple()
    {
        std::cout << "Destructing Simple" << m_nID << '\n';
    }

    int getID() { return m_nID; }
};

int main()
{
    // Allocate a Simple on the stack
    Simple simple{ 1 };
    std::cout << simple.getID() << '\n';

    // Allocate a Simple dynamically
    Simple* pSimple{ new Simple{ 2 } };

    std::cout << pSimple->getID() << '\n';

    // We allocated pSimple dynamically, so we have to delete it.
    delete pSimple;

    return 0;
} // simple goes out of scope here
```

This program produces the following result:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

A warning about the std::exit() function

Note that if you use the std::exit() function, your program will terminate and no destructors will be called. Be wary if you're relying on your destructors to do necessary cleanup work (e.g. write something to a log file or database before exiting).

19.4 Pointers to pointers

19.4.1 Declaration

A pointer to a pointer to an int is declared using two asterisks

```
int** ptrptr; // pointer to a pointer to an int, two asterisks
```

A pointer to a pointer works just like a normal pointer — you can dereference it to retrieve the value pointed to. And because that value is itself a pointer, you can dereference it again to get to the underlying value. These dereferences can be done consecutively:

```
int value { 5 };

int* ptr { &value };
std::cout << *ptr << '\n'; // Dereference pointer to int to get int value

int** ptrptr { &ptr };
std::cout << **ptrptr << '\n'; // dereference to get pointer to int,
    dereference again to get int value
```

The above program prints:

```
5
5
```

Note: you cannot set a pointer to a pointer directly to a value:

```
int value { 5 };
int** ptrptr { &&value }; // not valid
```

However, a pointer to a pointer can be set to null:

```
int** ptrptr { nullptr };
```

19.4.2 Arrays of pointers

Pointers to pointers have a few uses. The most common use is to dynamically allocate an array of pointers:

```
int** array { new int*[10] }; // allocate an array of 10 int pointers
```

This works just like a standard dynamically allocated array, except the array elements are of type "pointer to integer" instead of integer.

19.4.3 Two-dimensional dynamically allocated arrays

[...]

19.5 Void pointers

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type! A void pointer is declared like a normal pointer, using the void keyword as the pointer's type:

```cpp
void* ptr {}; // ptr is a void pointer
```

A void pointer can point to objects of any data type:

```cpp
int nValue {};
float fValue {};

struct Something
{
    int n;
    float f;
};

Something sValue {};

void* ptr {};
ptr = &nValue; // valid
ptr = &fValue; // valid
ptr = &sValue; // valid
```

However, because the void pointer does not know what type of object it is pointing to, dereferencing a void pointer is illegal. Instead, the void pointer must first be cast to another pointer type before the dereference can be performed.

```cpp
int value{ 5 };
void* voidPtr{ &value };

// std::cout << *voidPtr << '\n'; // illegal: dereference of void pointer

int* intPtr{ static_cast<int*>(voidPtr) }; // however, if we cast our void
    pointer to an int pointer...

std::cout << *intPtr << '\n'; // then we can dereference the result
```

The next obvious question is: If a void pointer doesn't know what it's pointing to, how do we know what to cast it to? Ultimately, that is up to you to keep track of.
Because a void pointer does not know what type of object it is pointing to, deleting a void pointer will result in undefined behavior. If you need to delete a void pointer, static_cast it back to the appropriate type first.
It is not possible to do pointer arithmetic on a void pointer. This is because pointer arithmetic requires the pointer to know what size object it is pointing to, so it can increment or decrement the pointer appropriately.
Note that there is no such thing as a void reference. This is because a void reference would be of type void &, and would not know what type of value it referenced.

Best practice: In general, it is a good idea to avoid using void pointers.

Chapter 20

Smart Pointers

20.1 Introduction to smart pointers and move semantics

https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/
Consider a function in which we dynamically allocate a value:

```cpp
void someFunction()
{
    Resource* ptr = new Resource(); // Resource is a struct or class

    // do stuff with ptr here

    delete ptr;
}
```

Although the above code seems fairly straightforward, it's fairly easy to forget to deallocate ptr. Even if you do remember to delete ptr at the end of the function, there are a myriad of ways that ptr may not be deleted if the function exits early. This can happen via an early return:

```cpp
#include <iostream>

void someFunction()
{
    Resource* ptr = new Resource();

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // the function returns early, and ptr won't be deleted!

    // do stuff with ptr here

    delete ptr;
}
```

or via a thrown exception...
At heart, these kinds of issues occur because pointer variables have no inherent mechanism to clean up after themselves.
One of the best things about classes is that they contain destructors that automatically get executed when an object of the class goes out of scope. So if you allocate (or acquire) memory in your constructor, you can deallocate it in your destructor, and be guaranteed that the memory will be deallocated when the class object is destroyed (regardless of whether it goes out of scope, gets explicitly deleted, etc. . .).
So can we use a class to help us manage and clean up our pointers? We can!
Consider a class whose sole job was to hold and "own" a pointer passed to it, and then deallocate that pointer when the class object went out of scope. As long as objects of that class were only created as local variables, we could guarantee that the class would properly go out of scope (regardless of when or how our functions terminate) and the owned pointer would get destroyed.
Here's a first draft of the idea:

```
#include <iostream>

template <typename T>
class Auto_ptr1
{
  T* m_ptr {};
public:
  // Pass in a pointer to "own" via the constructor
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  // The destructor will make sure it gets deallocated
  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  // Overload dereference and operator-> so we can use Auto_ptr1 like
     m_ptr.
  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

// A sample class to prove the above works
class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr1<Resource> res(new Resource()); // Note the allocation of
     memory here

        // ... but no explicit delete needed

  // Also note that we use <Resource>, not <Resource*>
        // This is because we've defined m_ptr to have type T* (not T)

  return 0;
} // res goes out of scope here, and destroys the allocated Resource for us
```

This program prints:

```
Resource acquired
Resource destroyed
```

Such a class is called a smart pointer. A **Smart pointer** is a composition class that is designed to manage dynamically allocated memory and ensure that memory gets deleted when the smart pointer object goes out of scope. (Relatedly, built-in pointers are sometimes called “dumb pointers” because they can't clean up after themselves).

A critical flaw

The `Auto_ptr1` class has a critical flaw lurking behind some auto-generated code. Consider the following program:

```
#include <iostream>

// Same as above
template <typename T>
class Auto_ptr1
```

```
{
  T* m_ptr {};
public:
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr1<Resource> res1(new Resource());
  Auto_ptr1<Resource> res2(res1); // Alternatively, don't initialize res2
      and then assign res2 = res1;

  return 0;
} // res1 and res2 go out of scope here
```

This program prints:

```
Resource acquired
Resource destroyed
Resource destroyed
```

Very likely (but not necessarily) your program will crash at this point. See the problem now? Because we haven't supplied a copy constructor or an assignment operator, C++ provides one for us. And the functions it provides do shallow copies. So when we initialize res2 with res1, both `Auto_ptr1` variables are pointed at the same Resource. When res2 goes out of the scope, it deletes the resource, leaving res1 with a dangling pointer. When res1 goes to delete its (already deleted) Resource, undefined behavior will result (probably a crash)!
You'd run into a similar problem with a function like this:

```
void passByValue(Auto_ptr1<Resource> res)
{
}

int main()
{
  Auto_ptr1<Resource> res1(new Resource());
  passByValue(res1);

  return 0;
}
In this program, res1 will be copied by value into parameter res, so both
    res1.m_ptr and res.m_ptr will hold the same address.
```

When res is destroyed at the end of the function, res1.m_ptr is left dangling. When res1.m_ptr is later deleted, undefined behavior will result.
So clearly this isn't good. How can we address this?
Well, one thing we could do would be to explicitly define and delete the copy constructor and assignment operator, thereby preventing any copies from being made in the first place. That would prevent the pass by value case (which is good, we probably shouldn't be passing these by value anyway).
But then how would we return an `Auto_ptr1` from a function back to the caller?

```
??? generateResource()
{
     Resource* r{ new Resource() };
     return Auto_ptr1(r);
}
```

We can't return our `Auto_ptr1` by reference, because the local `Auto_ptr1` will be destroyed at the end of the function, and the caller will be left with a dangling reference. We could return pointer r as Resource*, but then we might forget to delete r later, which is the whole point of using smart pointers in the first place. So that's out. Returning the `Auto_ptr1` by value is the only option that makes sense – but then we end up with shallow copies, duplicated pointers, and crashes.
Another option would be to overload the copy constructor and assignment operator to make deep copies. In this way, we'd at least guarantee to avoid duplicate pointers to the same object. But copying can be expensive (and may not be desirable or even possible), and we don't want to make needless copies of objects just to return an `Auto_ptr1` from a function. Plus assigning or initializing a dumb pointer doesn't copy the object being pointed to, so why would we expect smart pointers to behave differently? What do we do?

20.1.1 Move semantics

What if, instead of having our copy constructor and assignment operator copy the pointer ("copy semantics"), we instead transfer/move ownership of the pointer from the source to the destination object? This is the core idea behind move semantics. **Move semantics** means the class will transfer ownership of the object rather than making a copy.
Let's update our `Auto_ptr1` class to show how this can be done:

```
#include <iostream>

template <typename T>
class Auto_ptr2
{
  T* m_ptr {};
public:
  Auto_ptr2(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr2()
  {
    delete m_ptr;
  }

  // A copy constructor that implements move semantics
  Auto_ptr2(Auto_ptr2& a) // note: not const
  {
    // We don't need to delete m_ptr here.  This constructor is only
        called when we're creating a new object, and m_ptr can't be set
        prior to this.
    m_ptr = a.m_ptr; // transfer our dumb pointer from the source to our
        local object
    a.m_ptr = nullptr; // make sure the source no longer owns the pointer
  }

  // An assignment operator that implements move semantics
  Auto_ptr2& operator=(Auto_ptr2& a) // note: not const
  {
    if (&a == this)
      return *this;

    delete m_ptr; // make sure we deallocate any pointer the destination
        is already holding first
    m_ptr = a.m_ptr; // then transfer our dumb pointer from the source to
        the local object
    a.m_ptr = nullptr; // make sure the source no longer owns the pointer
    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr2<Resource> res1(new Resource());
  Auto_ptr2<Resource> res2; // Start as nullptr

  std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
  std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");

  res2 = res1; // res2 assumes ownership, res1 is set to null

  std::cout << "Ownership transferred\n";

  std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
  std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");

  return 0;
}
```

This program prints:

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

Yes Yes Non-modifiable l-values No No R-values No No
L-value reference to const Can be initialized with Can modify Modifiable l-values Yes No Non-modifiable l-values Yes No R-values Yes No

20.2 R-value references

https://www.learncpp.com/cpp-tutorial/rvalue-references/

20.2.1 L-value references recap

Prior to C++11, only one type of reference existed in C++, and so it was just called a "reference". However, in C++11, it's called an l-value reference. L-value references can only be initialized with modifiable l-values.

L-value reference	Can be initialized with	Can modify
Modifiable l-values	Yes	Yes
Non-modifiable l-values	No	No
R-values	No	No

L-value references to const objects can be initialized with modifiable and non-modifiable l-values and r-values alike. However, those values can't be modified.

L-value reference to const	Can be initialized with	Can modify
Modifiable l-values	Yes	No
Non-modifiable l-values	Yes	No
R-values	Yes	No

L-value references to const objects are particularly useful because they allow us to pass any type of argument (l-value or r-value) into a function without making a copy of the argument.

20.2.2 R-value references

Let's take a look at some examples:
C++11 adds a new type of reference called an r-value reference. An r-value reference is a reference that is designed to be initialized with an r-value (only). While an l-value reference is created using a single ampersand, an r-value reference is created using a double ampersand:

```cpp
int x{ 5 };
int& lref{ x }; // l-value reference initialized with l-value x
int&& rref{ 5 }; // r-value reference initialized with r-value 5
```

R-values references cannot be initialized with l-values.

R-value reference	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	Yes

R-value reference to const	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	No

R-value references have two properties that are useful. First, r-value references extend the lifespan of the object they are initialized with to the lifespan of the r-value reference (l-value references to const objects can do this too). Second, non-const r-value references allow you to modify the r-value!
Let's take a look at some examples:

```cpp
#include <iostream>

class Fraction
{
private:
  int m_numerator { 0 };
  int m_denominator { 1 };

public:
  Fraction(int numerator = 0, int denominator = 1) :
    m_numerator{ numerator }, m_denominator{ denominator }
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1)
  {
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
  }
};

int main()
{
  auto&& rref{ Fraction{ 3, 5 } }; // r-value reference to temporary
      Fraction

  // f1 of operator<< binds to the temporary, no copies are created.
  std::cout << rref << '\n';

  return 0;
} // rref (and the temporary Fraction) goes out of scope here
```

This program prints:

```
3/5
```

Now let's take a look at a less intuitive example:

```
#include <iostream>

int main()
{
    int&& rref{ 5 }; // because we're initializing an r-value reference
        with a literal, a temporary with value 5 is created here
    rref = 10;
    std::cout << rref << '\n';

    return 0;
}
```

This program prints:

```
10
```

R-value references are not very often used in either of the manners illustrated above.

20.2.3 R-value references as function parameters

R-value references are more often used as function parameters. This is most useful for function overloads when you want to have different behavior for l-value and r-value arguments.

```
#include <iostream>

void fun(const int& lref) // l-value arguments will select this function
{
  std::cout << "l-value reference to const: " << lref << '\n';
}

void fun(int&& rref) // r-value arguments will select this function
{
  std::cout << "r-value reference: " << rref << '\n';
}

int main()
{
  int x{ 5 };
  fun(x); // l-value argument calls l-value version of function
  fun(5); // r-value argument calls r-value version of function

  return 0;
}
```

Rvalue reference variables are lvalues

Consider the following snippet:

```
int&& ref{ 5 };
fun(ref);
Which version of fun would you expect the above to call: fun(const int&)
   or fun(int&&)?
```

The answer might surprise you. This calls fun(const int&).
Although variable ref has type int&&, when used in an expression it is an lvalue (as are all named variables). The type of an object and its value category are independent.

Best practice: You should almost never return an r-value reference, for the same reason you should almost never return an l-value reference. In most cases, you'll end up returning a hanging reference when the referenced object goes out of scope at the end of the function.

20.3 Move constructors and move assignment

`https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/`
First, let's take a moment to recap copy semantics: copy constructors and copy assignment.
Copy constructors are used to initialize a class by making a copy of an object of the same class. Copy assignment is used to copy one class object to another existing class object. By default, C++ will provide a copy constructor and copy assignment operator if one is not explicitly provided. These compiler-provided functions do shallow copies, which may cause problems for classes that allocate dynamic memory. So classes that deal with dynamic memory should override these functions to do deep copies.
C++11 defines two new functions in service of move semantics: a move constructor, and a move assignment operator. Whereas the goal of the copy constructor and copy assignment is to make a copy of one object to another, the goal of the move constructor and move assignment is to move ownership of the resources from one object to another (which is typically much less expensive than making a copy).

20.3.1 Move constructor

Defining a move constructor and move assignment work analogously to their copy counterparts. However, whereas the copy flavors of these functions take a const l-value reference parameter (which will bind to just about anything), the move flavors of these functions use non-const rvalue reference parameters (which only bind to rvalues).
Here's an example of the `Auto_ptr`4 class where we've left in the deep-copying copy constructor and copy assignment operator for comparison purposes.

```cpp
#include <iostream>

template<typename T>
class Auto_ptr4
{
  T* m_ptr {};
public:
  Auto_ptr4(T* ptr = nullptr)
    : m_ptr { ptr }
  {
  }

  ~Auto_ptr4()
  {
    delete m_ptr;
  }

  // Copy constructor
  // Do deep copy of a.m_ptr to m_ptr
  Auto_ptr4(const Auto_ptr4& a)
  {
    m_ptr = new T;
    *m_ptr = *a.m_ptr;
  }

  // Move constructor
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr4(Auto_ptr4&& a) noexcept
    : m_ptr { a.m_ptr }
  {
    a.m_ptr = nullptr; // we'll talk more about this line below
  }
```

```

	// Copy assignment
	// Do deep copy of a.m_ptr to m_ptr
	Auto_ptr4& operator=(const Auto_ptr4& a)
	{
		// Self-assignment detection
		if (&a == this)
			return *this;

		// Release any resource we're holding
		delete m_ptr;

		// Copy the resource
		m_ptr = new T;
		*m_ptr = *a.m_ptr;

		return *this;
	}

	// Move assignment
	// Transfer ownership of a.m_ptr to m_ptr
	Auto_ptr4& operator=(Auto_ptr4&& a) noexcept
	{
		// Self-assignment detection
		if (&a == this)
			return *this;

		// Release any resource we're holding
		delete m_ptr;

		// Transfer ownership of a.m_ptr to m_ptr
		m_ptr = a.m_ptr;
		a.m_ptr = nullptr; // we'll talk more about this line below

		return *this;
	}

	T& operator*() const { return *m_ptr; }
	T* operator->() const { return m_ptr; }
	bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
	Resource() { std::cout << "Resource acquired\n"; }
	~Resource() { std::cout << "Resource destroyed\n"; }
};

Auto_ptr4<Resource> generateResource()
{
	Auto_ptr4<Resource> res{new Resource};
	return res; // this return value will invoke the move constructor
}

int main()
{
	Auto_ptr4<Resource> mainres;
	mainres = generateResource(); // this assignment will invoke the move
	    assignment

	return 0;
}
```

The move constructor and move assignment operator are simple. Instead of deep copying the source object (a) into the destination object (the implicit object), we simply move (steal) the source object's resources. This

involves shallow copying the source pointer into the implicit object, then setting the source pointer to null. That's much better!
The flow of the program is exactly the same as before. However, instead of calling the copy constructor and copy assignment operators, this program calls the move constructor and move assignment operators. Looking a little more deeply:

- Inside generateResource(), local variable res is created and initialized with a dynamically allocated Resource, which causes the first "Resource acquired".
- Res is returned back to main() by value. Res is move constructed into a temporary object, transferring the dynamically created object stored in res to the temporary object. We'll talk about why this happens below.
- Res goes out of scope. Because res no longer manages a pointer (it was moved to the temporary), nothing interesting happens here.
- The temporary object is move assigned to mainres. This transfers the dynamically created object stored in the temporary to mainres.
- The assignment expression ends, and the temporary object goes out of expression scope and is destroyed. However, because the temporary no longer manages a pointer (it was moved to mainres), nothing interesting happens here either.
- At the end of main(), mainres goes out of scope, and our final "Resource destroyed" is displayed.

So instead of copying our Resource twice (once for the copy constructor and once for the copy assignment), we transfer it twice. This is more efficient, as Resource is only constructed and destroyed once instead of three times.

The key insight behind move semantics

You now have enough context to understand the key insight behind move semantics.
If we construct an object or do an assignment where the argument is an l-value, the only thing we can reasonably do is copy the l-value. We can't assume it's safe to alter the l-value, because it may be used again later in the program. If we have an expression "a = b" (where b is an lvalue), we wouldn't reasonably expect b to be changed in any way.
However, if we construct an object or do an assignment where the argument is an r-value, then we know that r-value is just a temporary object of some kind. Instead of copying it (which can be expensive), we can simply transfer its resources (which is cheap) to the object we're constructing or assigning. This is safe to do because the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again!
C++11, through r-value references, gives us the ability to provide different behaviors when the argument is an r-value vs an l-value, enabling us to make smarter and more efficient decisions about how our objects should behave.
Note: Move functions should always leave both objects in a valid state.
In the above examples, both the move constructor and move assignment functions set a.m_ptr to nullptr. This may seem extraneous – after all, if a is a temporary r-value, why bother doing "cleanup" if parameter a is going to be destroyed anyway?
The answer is simple: When a goes out of scope, the destructor for a will be called, and a.m_ptr will be deleted. If at that point, a.m_ptr is still pointing to the same object as m_ptr, then m_ptr will be left as a dangling pointer. When the object containing m_ptr eventually gets used (or destroyed), we'll get undefined behavior.
When implementing move semantics, it is important to ensure the moved-from object is left in a valid state, so that it will destruct properly (without creating undefined behavior).

20.3.2 Disabling copying

In the `Auto_ptr4` class above, we left in the copy constructor and assignment operator for comparison purposes. But in move-enabled classes, it is sometimes desirable to delete the copy constructor and copy assignment functions to ensure copies aren't made. In the case of our `Auto_ptr` class, we don't want to copy our templated object T – both because it's expensive, and whatever class T is may not even support copying!
Here's a version of `Auto_ptr` that supports move semantics but not copy semantics:

```
#include <iostream>

template<typename T>
```

```
class Auto_ptr5
{
  T* m_ptr {};
public:
  Auto_ptr5(T* ptr = nullptr)
    : m_ptr { ptr }
  {
  }

  ~Auto_ptr5()
  {
    delete m_ptr;
  }

  // Copy constructor -- no copying allowed!
  Auto_ptr5(const Auto_ptr5& a) = delete;

  // Move constructor
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr5(Auto_ptr5&& a) noexcept
    : m_ptr { a.m_ptr }
  {
    a.m_ptr = nullptr;
  }

  // Copy assignment -- no copying allowed!
  Auto_ptr5& operator=(const Auto_ptr5& a) = delete;

  // Move assignment
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr5& operator=(Auto_ptr5&& a) noexcept
  {
    // Self-assignment detection
    if (&a == this)
      return *this;

    // Release any resource we're holding
    delete m_ptr;

    // Transfer ownership of a.m_ptr to m_ptr
    m_ptr = a.m_ptr;
    a.m_ptr = nullptr;

    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};
```

If you were to try to pass an Auto_ptr5 l-value to a function by value, the compiler would complain that the copy constructor required to initialize the function parameter has been deleted. This is good, because we should probably be passing Auto_ptr5 by const l-value reference anyway!
Auto_ptr5 is (finally) a good smart pointer class. And, in fact the standard library contains a class very much like this one (that you should use instead), named std::unique_ptr. We'll talk more about std::unique_ptr later in this chapter.
Another example

20.4 `std::move`

`https://www.learncpp.com/cpp-tutorial/stdmove/`
In C++11, std::move is a standard library function that casts (using static_cast) its argument into an r-value reference, so that move semantics can be invoked. Thus, we can use std::move to cast an l-value into a type that will prefer being moved over being copied. std::move is defined in the utility header.

Here's the same program as above, but with a mySwapMove() function that uses std::move to convert our l-values into r-values so we can invoke move semantics:

```cpp
#include <iostream>
#include <string>
#include <utility> // for std::move

template <typename T>
void mySwapMove(T& a, T& b)
{
  T tmp { std::move(a) }; // invokes move constructor
  a = std::move(b); // invokes move assignment
  b = std::move(tmp); // invokes move assignment
}

int main()
{
  std::string x{ "abc" };
  std::string y{ "de" };

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  mySwapMove(x, y);

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  return 0;
}
```

Where else is std::move useful?
std::move can also be useful when sorting an array of elements. Many sorting algorithms (such as selection sort and bubble sort) work by swapping pairs of elements. In previous lessons, we've had to resort to copy-semantics to do the swapping. Now we can use move semantics, which is more efficient.
It can also be useful if we want to move the contents managed by one smart pointer to another.

20.5 std::unique_ptr

https://www.learncpp.com/cpp-tutorial/stdunique_ptr/
std::unique_ptr is the C++11 replacement for std::auto_ptr. It should be used to manage any dynamically allocated object that is not shared by multiple objects. That is, std::unique_ptr should completely own the object it manages, not share that ownership with other classes. std::unique_ptr lives in the ¡memory¿ header.
Let's take a look at a simple smart pointer example:

```cpp
#include <iostream>
#include <memory> // for std::unique_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // allocate a Resource object and have it owned by std::unique_ptr
  std::unique_ptr<Resource> res{ new Resource() };

  return 0;
} // res goes out of scope here, and the allocated Resource is destroyed
```

Remember that std::unique_ptr may not always be managing an object – either because it was created empty (using the default constructor or passing in a nullptr as the parameter), or because the resource it was managing

got moved to another std::unique_ptr. So before we use either of these operators, we should check whether the std::unique_ptr actually has a resource. Fortunately, this is easy: std::unique_ptr has a cast to bool that returns true if the std::unique_ptr is managing a resource.
Here's an example of this:

```cpp
#include <iostream>
#include <memory> // for std::unique_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

std::ostream& operator<<(std::ostream& out, const Resource&)
{
  out << "I am a resource";
  return out;
}

int main()
{
  std::unique_ptr<Resource> res{ new Resource{} };

  if (res) // use implicit cast to bool to ensure res contains a Resource
    std::cout << *res << '\n'; // print the Resource that res is owning

  return 0;
}
```

20.5.1 std::make_unique

C++14 comes with an additional function named std::make_unique(). This templated function constructs an object of the template type and initializes it with the arguments passed into the function.

```cpp
#include <memory> // for std::unique_ptr and std::make_unique
#include <iostream>

class Fraction
{
private:
  int m_numerator{ 0 };
  int m_denominator{ 1 };

public:
  Fraction(int numerator = 0, int denominator = 1) :
    m_numerator{ numerator }, m_denominator{ denominator }
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
  }
};


int main()
{
  // Create a single dynamically allocated Fraction with numerator 3 and
      denominator 5
  // We can also use automatic type deduction to good effect here
  auto f1{ std::make_unique<Fraction>(3, 5) };
  std::cout << *f1 << '\n';

  // Create a dynamically allocated array of Fractions of length 4
  auto f2{ std::make_unique<Fraction[]>(4) };
  std::cout << f2[0] << '\n';

  return 0;
}
```

20.5.2 Passing `std::unique_ptr` to a function

If you want the function to take ownership of the contents of the pointer, pass the std::unique_ptr by value. Note that because copy semantics have been disabled, you'll need to use std::move to actually pass the variable in.

```
#include <iostream>
#include <memory> // for std::unique_ptr
#include <utility> // for std::move

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

std::ostream& operator<<(std::ostream& out, const Resource&)
{
  out << "I am a resource";
  return out;
}

// This function takes ownership of the Resource, which isn't what we want
void takeOwnership(std::unique_ptr<Resource> res)
{
     if (res)
          std::cout << *res << '\n';
} // the Resource is destroyed here

int main()
{
    auto ptr{ std::make_unique<Resource>() };

//    takeOwnership(ptr); // This doesn't work, need to use move semantics
    takeOwnership(std::move(ptr)); // ok: use move semantics

    std::cout << "Ending program\n";

    return 0;
}
```

However, most of the time, you won't want the function to take ownership of the resource.
Although you can pass a std::unique_ptr by const reference (which will allow the function to use the object without assuming ownership), it's better to just pass the resource itself (by pointer or reference, depending on whether null is a valid argument). This allows the function to remain agnostic of how the caller is managing its resources.

`std::unique_ptr` and classes

You can, of course, use std::unique_ptr as a composition member of your class. This way, you don't have to worry about ensuring your class destructor deletes the dynamic memory, as the std::unique_ptr will be automatically destroyed when the class object is destroyed.
However, if the class object is not destroyed properly (e.g. it is dynamically allocated and not deallocated properly), then the std::unique_ptr member will not be destroyed either, and the object being managed by the std::unique_ptr will not be deallocated.

20.6 std::shared_ptr

https://www.learncpp.com/cpp-tutorial/stdshared_ptr/
Unlike std::unique_ptr, which is designed to singly own and manage a resource, `std::shared_ptr` is meant to solve the case where you need multiple smart pointers co-owning a resource.
This means that it is fine to have multiple `std::shared_ptr` pointing to the same resource. Internally, `std::shared_ptr` keeps track of how many `std::shared_ptr` are sharing the resource. As long as at least one `std::shared_ptr` is pointing to the resource, the resource will not be deallocated, even if individual `std::shared_ptr` are destroyed. As soon as the last `std::shared_ptr` managing the resource goes out of scope (or is reassigned to point at something else), the resource will be deallocated.
Like std::unique_ptr, `std::shared_ptr` lives in the ¡memory¿ header.

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // allocate a Resource object and have it owned by std::shared_ptr
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1{ res };
  {
    std::shared_ptr<Resource> ptr2 { ptr1 }; // make another
        std::shared_ptr pointing to the same thing

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, but nothing happens

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

Note that we created a second shared pointer from the first shared pointer. This is important. Consider the following similar program:

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1 { res };
  {
    std::shared_ptr<Resource> ptr2 { res }; // create ptr2 directly from
        res (instead of ptr1)

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, and the allocated Resource is destroyed

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
    again
```

This program prints and then crashes.
The difference here is that we created two **`std::shared_ptr`** independently from each other. As a consequence, even though they're both pointing to the same Resource, they aren't aware of each other. When ptr2 goes out of scope, it thinks it's the only owner of the Resource, and deallocates it. When ptr1 later goes out of the scope, it thinks the same thing, and tries to delete the Resource again. Then bad things happen.
Fortunately, this is easily avoided: if you need more than one **`std::shared_ptr`** to a given resource, copy an existing **`std::shared_ptr`**.

20.6.1 `std::make_shared`

Much like std::make_unique() can be used to create a std::unique_ptr in C++14, std::make_shared() can (and should) be used to make a **`std::shared_ptr`**. std::make_shared() is available in C++11.
Here's our original example, using std::make_shared():

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // allocate a Resource object and have it owned by std::shared_ptr
  auto ptr1 { std::make_shared<Resource>() };
  {
    auto ptr2 { ptr1 }; // create ptr2 using copy of ptr1

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, but nothing happens

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

Digging into `std::shared_ptr`

Unlike std::unique_ptr, which uses a single pointer internally, **`std::shared_ptr`** uses two pointers internally. One pointer points at the resource being managed. The other points at a "control block", which is a dynamically allocated object that tracks of a bunch of stuff, including how many **`std::shared_ptr`** are pointing at the resource. When a **`std::shared_ptr`** is created via a **`std::shared_ptr`** constructor, the memory for the managed object (which is usually passed in) and control block (which the constructor creates) are allocated separately. However, when using std::make_shared(), this can be optimized into a single memory allocation, which leads to better performance.
This also explains why independently creating two **`std::shared_ptr`** pointed to the same resource gets us into trouble. Each **`std::shared_ptr`** will have one pointer pointing at the resource. However, each **`std::shared_ptr`** will independently allocate its own control block, which will indicate that it is the only pointer owning that resource. Thus, when that **`std::shared_ptr`** goes out of scope, it will deallocate the resource, not realizing there are other **`std::shared_ptr`** also trying to manage that resource.
However, when a **`std::shared_ptr`** is cloned using copy assignment, the data in the control block can be appropriately updated to indicate that there are now additional **`std::shared_ptr`** co-managing the resource.
Conclusion: **`std::shared_ptr`** is designed for the case where you need multiple smart pointers co-managing the same resource. The resource will be deallocated when the last **`std::shared_ptr`** managing the resource is destroyed.

Chapter 21

Inheritance

21.1 Introduction to inheritance

https://www.learncpp.com/cpp-tutorial/introduction-to-inheritance/

21.1.1 Hierarchies

A hierarchy is a diagram that shows how various objects are related. Most hierarchies either show a progression over time (386 -¿ 486 -¿ Pentium), or categorize things in a way that moves from general to specific (fruit -¿ apple -¿ honeycrisp). If you've ever taken biology, the famous domain, kingdom, phylum, class, order, family, genus, and species ordering defines a hierarchy (from general to specific).
Another example of a hierarchy: a square is a rectangle, which is a quadrilateral, which is a shape. A right triangle is a triangle, which is also a shape.

21.2 Basic inheritance

https://www.learncpp.com/cpp-tutorial/basic-inheritance-in-c/
Now that we've talked about what inheritance is in an abstract sense, let's talk about how it's used within C++.
Inheritance in C++ takes place between classes. In an inheritance (is-a) relationship, the class being inherited from is called the **parent class**, **base class**, or **superclass**, and the class doing the inheriting is called the **child class**, **derived class**, or **subclass**.
A child class inherits both behaviors (member functions) and properties (member variables) from the parent (subject to some access restrictions that we'll cover in a future lesson). These variables and functions become members of the derived class.
Consider an example where we have a simple class to represent a generic person:

```cpp
#include <string>
#include <string_view>

class Person
{
// In this example, we're making our members public for simplicity
public:
    std::string m_name{};
    int m_age{};

    Person(std::string_view name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

Let's say we wanted to write a program that keeps track of information about some baseball players.

```
class BaseballPlayer
{
// In this example, we're making our members public for simplicity
public:
    double m_battingAverage{};
    int m_homeRuns{};

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage{battingAverage}, m_homeRuns{homeRuns}
    {
    }
};
```

Now, we also want to keep track of a baseball player's name and age, and we already have that information as part of our Person class.
We have three choices for how to add name and age to BaseballPlayer:

- Add name and age to the BaseballPlayer class directly as members. This is probably the worst choice, as we're duplicating code that already exists in our Person class. Any updates to Person will have to be made in BaseballPlayer too.
- Add Person as a member of BaseballPlayer using composition. But we have to ask ourselves, "does a BaseballPlayer have a Person"? No, it doesn't. So this isn't the right paradigm.
- Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an is-a relationship. Is a BaseballPlayer a Person? Yes, it is. So inheritance is a good choice here.

21.2.1 Making BaseballPlayer a derived class

To have BaseballPlayer inherit from our Person class, the syntax is fairly simple. After the class BaseballPlayer declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called public inheritance. We'll talk more about what public inheritance means in a future lesson.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage{};
    int m_homeRuns{};

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage{battingAverage}, m_homeRuns{homeRuns}
    {
    }
};
```

When BaseballPlayer inherits from Person, BaseballPlayer acquires the member functions and variables from Person. Additionally, BaseballPlayer defines two members of its own: m_battingAverage and m_homeRuns. This makes sense, since these properties are specific to a BaseballPlayer, not to any Person.

Inheritance chains

It's possible to inherit from a class that is itself derived from another class. There is nothing noteworthy or special when doing so – everything proceeds as in the examples above.

Why is this kind of inheritance useful?

Inheriting from a base class means we don't have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (e.g. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!
For example, if we ever added a new function to Person, then Employee, Supervisor, and BaseballPlayer would automatically gain access to it. If we added a new variable to Employee, then Supervisor would also gain access to it. This allows us to construct new classes in an easy, intuitive, and low-maintenance way!

21.3 Order of construction of derived classes

https://www.learncpp.com/cpp-tutorial/order-of-construction-of-derived-classes/
Consider this example:

21.3.1 Order of construction for inheritance chains

It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes.
C++ always constructs the "first" or "most base" class first. It then walks through the inheritance tree in order and constructs each successive derived class.
Here's a short program that illustrates the order of creation all along the inheritance chain.

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A\n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "B\n";
    }
};

class C: public B
{
public:
    C()
    {
        std::cout << "C\n";
    }
};

class D: public C
{
public:
    D()
    {
        std::cout << "D\n";
    }
};

int main()
{
    std::cout << "Constructing A: \n";
    A a;

    std::cout << "Constructing B: \n";
    B b;

    std::cout << "Constructing C: \n";
    C c;

    std::cout << "Constructing D: \n";
    D d;
}
```

This code prints the following:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

21.4 Constructors and initialization of derived classes

https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/

21.4.1 Initializing base class members

Consider there Base and Derived classes:

```cpp
class Base
{
public:
    int m_id {};

    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0)
        : m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the Base class constructor in the member initializer list of the derived class:

```cpp
class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0, int id=0)
        : Base{ id } // Call Base(int) constructor with value id!
        , m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

Now we can make our members private. Now that you know how to initialize base class members, there's no need to keep our member variables public. We make our member variables private again, as they should be.

21.4.2 Inheritance chains

```cpp
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A{ a }
    {
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a, double b, char c)
    : B{ a, b }
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c{ 5, 4.3, 'R' };

    return 0;
}
```

21.5 Inheritance and access specifiers

https://www.learncpp.com/cpp-tutorial/inheritance-and-access-specifiers/

21.5.1 The protected access specifier

C++ has a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The **protected** access specifier allows the class the member belongs to, friends, and derived classes to access the member. However, protected members are not accessible from outside the class.

```cpp
class Base
{
public:
    int m_public {}; // can be accessed by anybody
protected:
    int m_protected {}; // can be accessed by Base members, friends, and
        derived classes
private:
    int m_private {}; // can only be accessed by Base members and friends
        (but not derived classes)
};

class Derived: public Base
{
public:
    Derived()
    {
        m_public = 1; // allowed: can access public base members from
            derived class
        m_protected = 2; // allowed: can access protected base members
            from derived class
        m_private = 3; // not allowed: can not access private base members
            from derived class
    }
};

int main()
{
    Base base;
    base.m_public = 1; // allowed: can access public members from outside
        class
    base.m_protected = 2; // not allowed: can not access protected members
        from outside class
    base.m_private = 3; // not allowed: can not access private members
        from outside class

    return 0;
}
```

21.5.2 When to use protected

With a protected attribute in a base class, derived classes can access that member directly. This means that if you later change anything about that protected attribute (the type, what the value means, etc...), you'll probably need to change both the base class AND all of the derived classes.
Therefore, using the protected access specifier is most useful when you (or your team) are going to be the ones deriving from your own classes, and the number of derived classes is reasonable. That way, if you make a change to the implementation of the base class, and updates to the derived classes are necessary as a result, you can make the updates yourself (and have it not take forever, since the number of derived classes is limited).
Making your members private means the public and derived classes can't directly make changes to the base class. This is good for insulating the public or derived classes from implementation changes, and for ensuring invariants are maintained properly. However, it also means your class may need a larger public (or protected) interface to support all of the functions that the public or derived classes need for operation, which has its own cost to build, test, and maintain.
In general, it's better to make your members private if you can, and only use protected when derived classes are planned and the cost to build and maintain an interface to those private members is too high.

Best practice: Favor private members over protected members.

21.5.3 Different kinds of inheritance

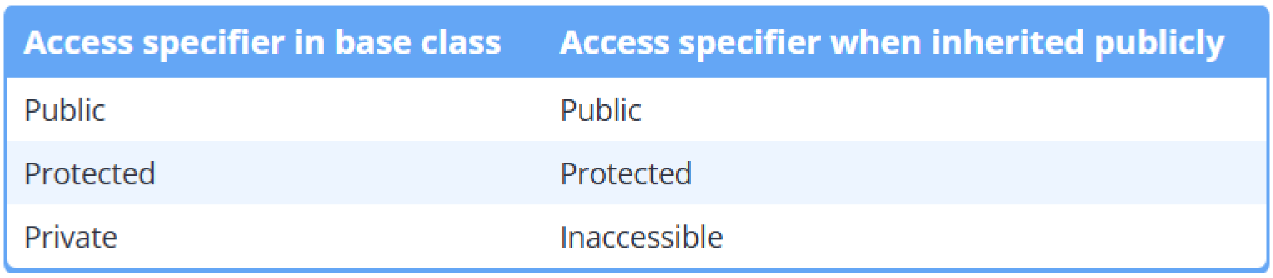
There are three different ways for classes to inherit from other classes: public, protected, and private.

```cpp
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).
So what's the difference between these? In a nutshell, when members are inherited, the access specifier for an inherited member may be changed (in the derived class only) depending on the type of inheritance used. Put another way, members that were public or protected in the base class may change access specifiers in the derived class.

21.5.4 Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you see or use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, inherited public members stay public, and inherited protected members stay protected. Inherited private members, which were inaccessible because they were private in the base class, stay inaccessible.



Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

Here's an example showing how things work:

```cpp
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};

class Pub: public Base // note: public inheritance
{
    // Public inheritance means:
    // Public inherited members stay public (so m_public is treated as
       public)
    // Protected inherited members stay protected (so m_protected is
       treated as protected)
    // Private inherited members stay inaccessible (so m_private is
       inaccessible)
public:
    Pub()
    {
        m_public = 1; // okay: m_public was inherited as public
        m_protected = 2; // okay: m_protected was inherited as protected
        m_private = 3; // not okay: m_private is inaccessible from derived
           class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being
        accessed.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pub pub;
    pub.m_public = 1; // okay: m_public is public in Pub
    pub.m_protected = 2; // not okay: m_protected is protected in Pub
    pub.m_private = 3; // not okay: m_private is inaccessible in Pub

    return 0;
}
```

Best practice: Use public inheritance unless you have a specific reason to do otherwise.

21.5.5 Protected inheritance

Protected inheritance is the least common method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay inaccessible.
Because this form of inheritance is so rare, we'll skip the example and just summarize with a table:

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

21.5.6 Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members are inaccessible, and protected and public members become private.
Note that this does not affect the way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

```
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};

class Pri: private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private (so m_public is treated as
        private)
    // Protected inherited members become private (so m_protected is
        treated as private)
```

```cpp
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pri()
    {
        m_public = 1; // okay: m_public is now private in Pri
        m_protected = 2; // okay: m_protected is now private in Pri
        m_private = 3; // not okay: derived classes can't access private members in the base class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of base.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pri pri;
    pri.m_public = 1; // not okay: m_public is now private in Pri
    pri.m_protected = 2; // not okay: m_protected is now private in Pri
    pri.m_private = 3; // not okay: m_private is inaccessible in Pri

    return 0;
}
```

To summarize in table form:

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Private inheritance can be useful when the derived class has no obvious relationship to the base class, but uses the base class for implementation internally. In such a case, we probably don't want the public interface of the base class to be exposed through objects of the derived class (as it would be if we inherited publicly).
In practice, private inheritance is rarely used.

21.6 New functionality to a derived class

https://www.learncpp.com/cpp-tutorial/adding-new-functionality-to-a-derived-class/
First, let's start with a simple base class:

```cpp
#include <iostream>

class Base
{
protected:
    int m_value {};

public:
    Base(int value)
        : m_value { value }
    {
    }

    void identify() const { std::cout << "I am a Base\n"; }
};
```

Now, let's create a derived class that inherits from Base. Because we want the derived class to be able to set the value of m_value when derived objects are instantiated, we'll make the Derived constructor call the Base constructor in the initialization list.

```cpp
class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }
};
```

21.6.1 Adding new functionality to a derived class

In the above example, because we have access to the source code of the Base class, we can add functionality directly to Base if we desire.
There may be times when we have access to a base class but do not want to modify it. Consider the case where you have just purchased a library of code from a 3rd party vendor, but need some extra functionality. You could add to the original code, but this isn't the best solution. What if the vendor sends you an update? Either your additions will be overwritten, or you'll have to manually migrate them into the update, which is time-consuming and risky.
Alternatively, there may be times when it's not even possible to modify the base class. Consider the code in the standard library. We aren't able to modify the code that's part of the standard library. But we are able to inherit from those classes, and then add our own functionality into our derived classes. The same goes for 3rd party libraries where you are provided with headers but the code comes precompiled.
In either case, the best answer is to derive your own class, and add the functionality you want to the derived class.
One obvious omission from the Base class is a way for the public to access m_value. We could remedy this by adding an access function in the Base class – but for the sake of example we're going to add it to the derived class instead. Because m_value has been declared as protected in the Base class, Derived has direct access to it.
To add new functionality to a derived class, simply declare that functionality in the derived class like normal:

```cpp
class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    int getValue() const { return m_value; }
};
```

21.7 Calling inherited functions and overriding behavior

https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/
By default, derived classes inherit all of the behaviors defined in a base class. In this lesson, we'll examine in more detail how member functions are selected, as well as how we can leverage this to change behaviors in a derived class.
When a member function is called on a derived class object, the compiler first looks to see if any function with that name exists in the derived class. If so, all overloaded functions with that name are considered, and the function overload resolution process is used to determine whether there is a best match. If not, the compiler walks up the inheritance chain, checking each parent class in turn in the same way.
Put another way, the compiler will select the best matching function from the most-derived class with at least one function with that name.

21.7.1 Calling a base class function

First, let's explore what happens when the derived class has no matching function, but the base class does:

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

    void identify() const { std::cout << "Base::identify()\n"; }
};

class Derived: public Base
{
public:
    Derived() { }
};

int main()
{
    Base base {};
    base.identify();

    Derived derived {};
    derived.identify();

    return 0;
}
```

This prints:

```
Base::identify()
Base::identify()
```

When base.identify() is called, the compiler looks to see if a function named identify() has been defined in class Base. It has, so the compiler looks to see if it is a match. It is, so it is called
When derived.identify() is called, the compiler looks to see if a function named identify() has been defined in the Derived class. It hasn't. So it moves to the parent class (in this case, Base), and tries again there. Base has defined an identify() function, so it uses that one. In other words, Base::identify() was used because Derived::identify() doesn't exist.
This means that if the behavior provided by a base class is sufficient, we can simply use the base class behavior.

21.7.2 Redefining behaviors

However, if we had defined Derived::identify() in the Derived class, it would have been used instead.
This means that we can make functions work differently with our derived classes by redefining them in the derived class!
To modify the way a function defined in a base class works in the derived class, simply redefine the function in the derived class.

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

    void identify() const { std::cout << "Base::identify()\n"; }
};

class Derived: public Base
{
public:
    Derived() { }

    void identify() const { std::cout << "Derived::identify()\n"; }
};

int main()
{
    Base base {};
    base.identify();

    Derived derived {};
    derived.identify();

    return 0;
}
```

Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can be redefined as public in the derived class, or vice-versa!

21.7.3 Adding to existing functionality

Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it when called with a derived object. In the above example, note that Derived::identify() completely hides Base::identify()! This may not be what we want. It is possible to have our derived function call the base version of the function of the same name (in order to reuse code) and then add additional functionality to it.
To have a derived function call a base function of the same name, simply do a normal function call, but prefix the function with the scope qualifier of the base class. For example:ù

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

    void identify() const { std::cout << "Base::identify()\n"; }
};

class Derived: public Base
{
public:
    Derived() { }

    void identify() const
    {
        std::cout << "Derived::identify()\n";
        Base::identify(); // note call to Base::identify() here
    }
};

int main()
{
    Base base {};
    base.identify();

    Derived derived {};
    derived.identify();

    return 0;
}
```

There's one bit of trickiness that we can run into when trying to call friend functions in base classes, such as operator¡¡. Because friend functions of the base class aren't actually part of the base class, using the scope resolution qualifier won't work. Instead, we need a way to make our Derived class temporarily look like the Base class so that the right version of the function can be called.

Fortunately, that's easy to do, using static_cast. Here's an example:

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

  friend std::ostream& operator<< (std::ostream& out, const Base&)
  {
    out << "In Base\n";
    return out;
  }
};

class Derived: public Base
{
public:
    Derived() { }

  friend std::ostream& operator<< (std::ostream& out, const Derived& d)
  {
    out << "In Derived\n";
    // static_cast Derived to a Base object, so we call the right version
        of operator<<
    out << static_cast<const Base&>(d);
    return out;
    }
};

int main()
{
    Derived derived {};

    std::cout << derived << '\n';

    return 0;
}
```

21.7.4 Overload resolution in derived classes

As noted at the top of the lesson, the compiler will select the best matching function from the most-derived class with at least one function with that name.
A using-declaration in Derived makes all Base functions with a certain name visible from within Derived:

```cpp
#include <iostream>

class Base
{
public:
    void print(int)    { std::cout << "Base::print(int)\n"; }
    void print(double) { std::cout << "Base::print(double)\n"; }
};

class Derived: public Base
{
public:
    using Base::print; // make all Base::print() functions eligible for
        overload resolution
    void print(double) { std::cout << "Derived::print(double)"; }
};


int main()
{
    Derived d{};
    d.print(5); // calls Base::print(int), which is the best matching
        function visible in Derived

    return 0;
}
```

By putting the using-declaration using Base::print; inside Derived, we are telling the compiler that all Base functions named print should be visible in Derived, which will cause them to be eligible for overload resolution. As a result, Base::print(int) is selected over Derived::print(double) when called on a int.

21.8 Hiding inherited functionality

https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/

21.8.1 Changing an inherited member's access level

C++ gives us the ability to change an inherited member's access specifier in the derived class. This is done by using a using declaration to identify the (scoped) base class member that is having its access changed in the derived class, under the new access specifier.
For example, consider the following Base:

```
#include <iostream>

class Base
{
private:
    int m_value {};

public:
    Base(int value)
        : m_value { value }
    {
    }

protected:
    void printValue() const { std::cout << m_value; }
};
```

Because Base::printValue() has been declared as protected, it can only be called by Base or its derived classes. The public can not access it.
Let's define a Derived class that changes the access specifier of printValue() to public:

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    // Base::printValue was inherited as protected, so the public has no
        access
    // But we're changing it to public via a using declaration
    using Base::printValue; // note: no parenthesis here
};
```

This means that this code will now work:

```
int main()
{
    Derived derived { 7 };

    // printValue is public in Derived, so this is okay
    derived.printValue(); // prints 7
    return 0;
}
```

You can only change the access specifiers of base members the derived class would normally be able to access. Therefore, you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

21.8.2 Hiding functionality

In C++, it is not possible to remove or restrict functionality from a base class other than by modifying the source code. However, in a derived class, it is possible to hide functionality that exists in the base class, so that it can not be accessed through the derived class. This can be done simply by changing the relevant access specifier.
For example, we can make a public member private:

```
#include <iostream>

class Base
{
public:
  int m_value{};
};

class Derived : public Base
{
private:
  using Base::m_value;

public:
  Derived(int value) : Base { value }
  {
  }
};

int main()
{
  Derived derived{ 7 };
  std::cout << derived.m_value; // error: m_value is private in Derived

  Base& base{ derived };
  std::cout << base.m_value; // okay: m_value is public in Base

  return 0;
}
```

21.8.3 Deleting functions in the derived class

You can also mark member functions as deleted in the derived class, which ensures they can't be called at all through a derived object:

```
#include <iostream>
class Base
{
private:
  int m_value {};

public:
  Base(int value)
    : m_value { value }
  {
  }

  int getValue() const { return m_value; }
};

class Derived : public Base
```

```cpp
{
public:
  Derived(int value)
    : Base { value }
  {
  }


  int getValue() const = delete; // mark this function as inaccessible
};

int main()
{
  Derived derived { 7 };

  // The following won't work because getValue() has been deleted!
  std::cout << derived.getValue();

  return 0;
}
```

21.9 Multiple inheritance

https://www.learncpp.com/cpp-tutorial/multiple-inheritance/
So far, all of the examples of inheritance we've presented have been single inheritance – that is, each inherited class has one and only one parent. However, C++ provides the ability to do multiple inheritance. **Multiple inheritance** enables a derived class to inherit members from more than one parent.
Let's say we wanted to write a program to keep track of a bunch of teachers. A teacher is a person. However, a teacher is also an employee (they are their own employer if working for themselves). Multiple inheritance can be used to create a Teacher class that inherits properties from both Person and Employee. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma.

```cpp
#include <string>
#include <string_view>

class Person
{
private:
    std::string m_name{};
    int m_age{};

public:
    Person(std::string_view name, int age)
        : m_name{ name }, m_age{ age }
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }
};

class Employee
{
private:
    std::string m_employer{};
    double m_wage{};

public:
    Employee(std::string_view employer, double wage)
        : m_employer{ employer }, m_wage{ wage }
    {
    }

    const std::string& getEmployer() const { return m_employer; }
    double getWage() const { return m_wage; }
};

// Teacher publicly inherits Person and Employee
class Teacher : public Person, public Employee
{
private:
    int m_teachesGrade{};

public:
    Teacher(std::string_view name, int age, std::string_view employer,
        double wage, int teachesGrade)
        : Person{ name, age }, Employee{ employer, wage }, m_teachesGrade{
            teachesGrade }
    {
    }
};

int main()
{
    Teacher t{ "Mary", 45, "Boo", 14.3, 8 };

    return 0;
}
```

21.9.1 Mixins

[...]

21.9.2 Problems with multiple inheritance

While multiple inheritance seems like a simple extension of single inheritance, multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare. Let's take a look at some of these situations.
[...]

Best practice: Avoid multiple inheritance unless alternatives lead to more complexity.

21.10 Pointers and references to the base class of derived objects

https://www.learncpp.com/cpp-tutorial/pointers-and-references-to-the-base-class/
In the previous chapter, you learned all about how to use inheritance to derive new classes from existing classes. In this chapter, we are going to focus on one of the most important and powerful aspects of inheritance – virtual functions.
But before we discuss what virtual functions are, let's first set the table for why we need them.
In the chapter on construction of derived classes, you learned that when you create a derived class, it is composed of multiple parts: one part for each inherited class, and a part for itself.

21.10.1 Pointers, references, and derived classes

It should be fairly intuitive that we can set Derived pointers and references to Derived objects:

```
#include <iostream>

int main()
{
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value "
        << derived.getValue() << '\n';

    Derived& rDerived{ derived };
```

```cpp
    std::cout << "rDerived is a " << rDerived.getName() << " and has value
        " << rDerived.getValue() << '\n';

    Derived* pDerived{ &derived };
    std::cout << "pDerived is a " << pDerived->getName() << " and has
        value " << pDerived->getValue() << '\n';

    return 0;
}
```

This produces the following output:

```
derived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

However, since Derived has a Base part, a more interesting question is whether C++ will let us set a Base pointer or reference to a Derived object. It turns out, we can!

```cpp
#include <iostream>

int main()
{
    Derived derived{ 5 };

    // These are both legal!
    Base& rBase{ derived }; // rBase is an lvalue reference (not an rvalue
        reference)
    Base* pBase{ &derived };

    std::cout << "derived is a " << derived.getName() << " and has value "
        << derived.getValue() << '\n';
    std::cout << "rBase is a " << rBase.getName() << " and has value " <<
        rBase.getValue() << '\n';
    std::cout << "pBase is a " << pBase->getName() << " and has value " <<
        pBase->getValue() << '\n';

    return 0;
}
```

This result may not be quite what you were expecting at first!
This produces the result:

```
derived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited). So even though Derived::getName() shadows (hides) Base::getName() for Derived objects, the Base pointer/reference can not see Derived::getName(). Consequently, they call Base::getName(), which is why rBase and pBase report that they are a Base rather than a Derived.
Note that this also means it is not possible to call Derived::getValueDoubled() using rBase or pBase. They are unable to see anything in Derived.

21.10.2 Use for pointers and references to base classes

Now you might be saying, “The above examples seem kind of silly. Why would I set a pointer or reference to the base class of a derived object when I can just use the derived object?” It turns out that there are quite a few good reasons.
First, let’s say you wanted to write a function that printed an animal’s name and sound. Without using a pointer to a base class, you’d have to write it using overloaded functions, like this:

```cpp
void report(const Cat& cat)
{
    std::cout << cat.getName() << " says " << cat.speak() << '\n';
}

```

```cpp
void report(const Dog& dog)
{
    std::cout << dog.getName() << " says " << dog.speak() << '\n';
}
```

Not too difficult, but consider what would happen if we had 30 different animal types instead of 2. You'd have to write 30 almost identical functions! Plus, if you ever added a new type of animal, you'd have to write a new function for that one too. This is a huge waste of time considering the only real difference is the type of the parameter.
However, because Cat and Dog are derived from Animal, Cat and Dog have an Animal part. Therefore, it makes sense that we should be able to do something like this:

```cpp
void report(const Animal& rAnimal)
{
    std::cout << rAnimal.getName() << " says " << rAnimal.speak() << '\n';
}
```

This would let us pass in any class derived from Animal, even ones that we created after we wrote the function! Instead of one function per derived class, we get one function that works with all classes derived from Animal!
The problem is, of course, that because rAnimal is an Animal reference, rAnimal.speak() will call Animal::speak() instead of the derived version of speak().
Second, let's say you had 3 cats and 3 dogs that you wanted to keep in an array for easy access. Because arrays can only hold objects of one type, without a pointer or reference to a base class, you'd have to create a different array for each derived type, like this:

```cpp
    const auto& cats{ std::to_array<Cat>({{ "Fred" }, { "Misty" }, {
        "Zeke" }}) };
    const auto& dogs{ std::to_array<Dog>({{ "Garbo" }, { "Pooky" }, {
        "Truffle" }}) };
```

Now, consider what would happen if you had 30 different types of animals. You'd need 30 arrays, one for each type of animal!
However, because both Cat and Dog are derived from Animal, it makes sense that we should be able to do something like this:

```cpp
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const Cat fred{ "Fred" };
    const Cat misty{ "Misty" };
    const Cat zeke{ "Zeke" };

    const Dog garbo{ "Garbo" };
    const Dog pooky{ "Pooky" };
    const Dog truffle{ "Truffle" };

    // Set up an array of pointers to animals, and set those pointers to
        our Cat and Dog objects
    const auto animals{ std::to_array<const Animal*>({&fred, &garbo,
        &misty, &pooky, &truffle, &zeke }) };

    // Before C++20, with the array size being explicitly specified
    // const std::array<const Animal*, 6> animals{ &fred, &garbo, &misty,
        &pooky, &truffle, &zeke };

    for (const auto animal : animals)
    {
        std::cout << animal->getName() << " says " << animal->speak() <<
            '\n';
    }

    return 0;
}
```

While this compiles and executes, unfortunately the fact that each element of array "animals" is a pointer to an Animal means that animal-¿speak() will call Animal::speak() instead of the derived class version of speak() that we want. The output is:

```
Fred says ???
Garbo says ???
Misty says ???
Pooky says ???
Truffle says ???
Zeke says ???
```

21.11 Virtual functions and polymorphism

https://www.learncpp.com/cpp-tutorial/virtual-functions/

21.11.1 Virtual functions

A **virtual function** is a special type of member function that, when called, resolves to the most-derived version of the function for the actual type of the object being referenced or pointed to.
A derived function is considered a match if it has the same signature (name, parameter types, and whether it is const) and return type as the base version of the function. Such functions are called **overrides**.
To make a function virtual, simply place the "virtual" keyword before the function declaration.
Let's take a look at an example:

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

int main()
{
    C c {};
    A& rBase{ c };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

As a result, our program outputs:

```
rBase is a C
```

Note that virtual function resolution only works when a virtual member function is called through a pointer or reference to a class type object. This works because the compiler can differentiate the type of the pointer or reference from the type of the object being pointed to or referenced. We see this in example above.
Calling a virtual member function directly on an object (not through a pointer or reference) will always invoke the member function belonging to the same type of that object. For example:

```cpp
C c{};
std::cout << c.getName(); // will always call C::getName

A a { c }; // copies the A portion of c into a (don't do this)
std::cout << a.getName(); // will always call A::getName
```

21.11.2 Polymorphism

In programming, **polymorphism** refers to the ability of an entity to have multiple forms (the term "polymorphism" literally means "many forms"). For example, consider the following two function declarations:

```cpp
int add(int, int);
double add(double, double);
```

The identifier add has two forms: add(int, int) and add(double, double).
Compile-time polymorphism refers to forms of polymorphism that are resolved by the compiler. These include function overload resolution, as well as template resolution.
Runtime polymorphism refers to forms of polymorphism that are resolved at runtime. This includes virtual function resolution.

21.11.3 Return types of virtual functions

Under normal circumstances, the return type of a virtual function and its override must match. Consider the following example:

```cpp
class Base
{
public:
    virtual int getValue() const { return 5; }
};

class Derived: public Base
{
public:
    virtual double getValue() const { return 6.78; }
};
```

In this case, Derived::getValue() is not considered a matching override for Base::getValue() and compilation will fail.

Do not call virtual functions from constructors or destructors

Here's another gotcha that often catches unsuspecting new programmers. You should not call virtual functions from constructors or destructors. Why?
Remember that when a Derived class is created, the Base portion is constructed first. If you were to call a virtual function from the Base constructor, and Derived portion of the class hadn't even been created yet, it would be unable to call the Derived version of the function because there's no Derived object for the Derived function to work on. In C++, it will call the Base version instead.
A similar issue exists for destructors. If you call a virtual function in a Base class destructor, it will always resolve to the Base class version of the function, because the Derived portion of the class will already have been destroyed.

21.11.4 The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient – resolving a virtual function call takes longer than resolving a regular one.
Furthermore, to make virtual functions work, the compiler has to allocate an extra pointer for each object of a class that has virtual functions. This adds a lot of overhead to objects that otherwise have a small size. We'll talk about this more in future lessons in this chapter.

21.12 Override and final specifiers

https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers/

21.12.1 The override specifier

As we mentioned in the previous lesson, a derived class virtual function is only considered an override if its signature and return types match exactly. That can lead to inadvertent issues, where a function that was intended to be an override actually isn't.
Consider the following example:

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
  virtual std::string_view getName1(int x) { return "A"; }
  virtual std::string_view getName2(int x) { return "A"; }
};

class B : public A
{
public:
  virtual std::string_view getName1(short x) { return "B"; } // note:
      parameter is a short
  virtual std::string_view getName2(int x) const { return "B"; } // note:
      function is const
};

int main()
{
  B b{};
  A& rBase{ b };
  std::cout << rBase.getName1(1) << '\n';
  std::cout << rBase.getName2(2) << '\n';

  return 0;
}
```

Because rBase is an A reference to a B object, the intention here is to use virtual functions to access B::getName1() and B::getName2(). However, because B::getName1() takes a different parameter (a short instead of an int), it's not considered an override of A::getName1(). More insidiously, because B::getName2() is const and A::getName2() isn't, B::getName2() isn't considered an override of A::getName2().
Consequently, this program prints:

```
A
A
```

To help address the issue of functions that are meant to be overrides but aren't, the override specifier can be applied to any virtual function to tell the compiler to enforce that the function is an override. The override specifier is placed at the end of a member function declaration (in the same place where a function-level const goes). If a member function is const and an override, the const must come before override.
If a function marked as override does not override a base class function (or is applied to a non-virtual function), the compiler will flag the function as an error.

```cpp
#include <string_view>

class A
{
public:
  virtual std::string_view getName1(int x) { return "A"; }
  virtual std::string_view getName2(int x) { return "A"; }
  virtual std::string_view getName3(int x) { return "A"; }
};

```

```cpp
class B : public A
{
public:
  std::string_view getName1(short int x) override { return "B"; } // compile error, function is not an override
  std::string_view getName2(int x) const override { return "B"; } // compile error, function is not an override
  std::string_view getName3(int x) override { return "B"; } // okay, function is an override of A::getName3(int)

};

int main()
{
  return 0;
}
```

Because there is no performance penalty for using the override specifier and it helps ensure you've actually overridden the function you think you have, all virtual override functions should be tagged using the override specifier. Additionally, because the override specifier implies virtual, there's no need to tag functions using the override specifier with the virtual keyword.

Best practice: Use the virtual keyword on virtual functions in a base class.

Use the override specifier (but not the virtual keyword) on override functions in derived classes. This includes virtual destructors.

Rule: If a member function is both const and an override, the const must be listed first. const override is correct, override const is not.

21.12.2 The final specifier

There may be cases where you don't want someone to be able to override a virtual function, or inherit from a class. The final specifier can be used to tell the compiler to enforce this. If the user tries to override a function or inherit from a class that has been specified as final, the compiler will give a compile error.
In the case where we want to restrict the user from overriding a function, the **final** specifier is used in the same place the override specifier is, like so:

```cpp
#include <string_view>

class A
{
public:
  virtual std::string_view getName() const { return "A"; }
};

class B : public A
{
public:
  // note use of final specifier on following line -- that makes this function not able to be overridden in derived classes
  std::string_view getName() const override final { return "B"; } // okay, overrides A::getName()
};

class C : public B
{
public:
  std::string_view getName() const override { return "C"; } // compile error: overrides B::getName(), which is final
};
```

In the case where we want to prevent inheriting from a class, the final specifier is applied after the class name:

```cpp
#include <string_view>

class A
{
public:
  virtual std::string_view getName() const { return "A"; }
};

class B final : public A // note use of final specifier here
{
public:
  std::string_view getName() const override { return "B"; }
};

class C : public B // compile error: cannot inherit from final class
{
public:
  std::string_view getName() const override { return "C"; }
};
```

In the above example, class B is declared final. Thus, when C tries to inherit from B, the compiler will give a compile error.

21.12.3 Covariant return types

There is one special case in which a derived class virtual function override can have a different return type than the base class and still be considered a matching override. If the return type of a virtual function is a pointer or a reference to some class, override functions can return a pointer or a reference to a derived class. These are called **covariant return types**. Here is an example:

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
  // This version of getThis() returns a pointer to a Base class
  virtual Base* getThis() { std::cout << "called Base::getThis()\n";
      return this; }
  void printType() { std::cout << "returned a Base\n"; }
};

class Derived : public Base
{
public:
  // Normally override functions have to return objects of the same type
      as the base function
  // However, because Derived is derived from Base, it's okay to return
      Derived* instead of Base*
  Derived* getThis() override { std::cout << "called
      Derived::getThis()\n";  return this; }
  void printType() { std::cout << "returned a Derived\n"; }
};

int main()
{
  Derived d{};
  Base* b{ &d };
  d.getThis()->printType(); // calls Derived::getThis(), returns a
      Derived*, calls Derived::printType
  b->getThis()->printType(); // calls Derived::getThis(), returns a Base*,
      calls Base::printType

  return 0;
}
```

This prints:

```
called Derived::getThis()
returned a Derived
called Derived::getThis()
returned a Base
```

21.13 Virtual destructors, virtual assignment, and overriding virtualization

https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment/

21.13.1 Virtual destructors

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should always make your destructors virtual if you're dealing with inheritance.
Consider the following example:

```cpp
#include <iostream>
class Base
{
public:
    ~Base() // note: not virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    ~Derived() // note: not virtual (your compiler may warn you about this)
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Because base is a Base pointer, when base is deleted, the program looks to see if the Base destructor is virtual. It's not, so it assumes it only needs to call the Base destructor. We can see this in the fact that the above example prints:

```
Calling ~Base()
```

However, we really want the delete function to call Derived's destructor (which will call Base's destructor in turn), otherwise m_array will not be deleted. We do this by making Base's destructor virtual:

```cpp
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Now this program produces the following result:

```
Calling ~Derived()
Calling ~Base()
```

Rule: Whenever you are dealing with inheritance, you should make any explicit destructors virtual.

As with normal virtual member functions, if a base class function is virtual, all derived overrides will be considered virtual regardless of whether they are specified as such. It is not necessary to create an empty derived class destructor just to mark it as virtual.
Note that if you want your base class to have a virtual destructor that is otherwise empty, you can define your destructor this way:

```cpp
virtual ~Base() = default; // generate a virtual default destructor
```

21.13.2 Virtual assignment

It is possible to make the assignment operator virtual. However, unlike the destructor case where virtualization is always a good idea, virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial. Consequently, we are going to recommend you leave your assignments non-virtual for now, in the interest of simplicity.

21.13.3 Ignoring virtualization

Very rarely you may want to ignore the virtualization of a function. For example, consider the following code:

```cpp
#include <string_view>
class Base
{
public:
    virtual ~Base() = default;
    virtual std::string_view getName() const { return "Base"; }
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};
```

There may be cases where you want a Base pointer to a Derived object to call Base::getName() instead of Derived::getName(). To do so, simply use the scope resolution operator:

```cpp
#include <iostream>
int main()
{
    Derived derived {};
    const Base& base { derived };

    // Calls Base::getName() instead of the virtualized Derived::getName()
    std::cout << base.Base::getName() << '\n';

    return 0;
}
```

Should we make all destructors virtual?

This is a common question asked by new programmers. As noted in the top example, if the base class destructor isn't marked as virtual, then the program is at risk for leaking memory if a programmer later deletes a base class pointer that is pointing to a derived object. One way to avoid this is to mark all your destructors as virtual. But should you?
It's easy to say yes, so that way you can later use any class as a base class – but there's a performance penalty for doing so (a virtual pointer added to every instance of your class). So you have to balance that cost, as well as your intent.

21.14 Early binding and late binding

`https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/`
In this lesson and the next, we are going to take a closer look at how virtual functions are implemented. While this information is not strictly necessary to effectively use virtual functions, it is interesting. Nevertheless, you can consider both sections optional reading.
[...]

21.15 The virtual table

`https://www.learncpp.com/cpp-tutorial/the-virtual-table/`
The C++ standard does not specify how virtual functions should be implemented (this detail is left up to the implementation).
However, C++ implementations typically implement virtual functions using a form of late binding known as the virtual table.
The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as "vtable", "virtual function table", "virtual method table", or "dispatch table". In C++, virtual function resolution is sometimes called **dynamic dispatch**.

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.
[...]

21.16 Pure virtual functions, abstract base classes, and interface classes

https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes/

21.16.1 Pure virtual (abstract) functions and abstract base classes

So far, all of the virtual functions we have written have a body (a definition). However, C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```cpp
#include <string_view>

class Base
{
public:
    std::string_view sayHi() const { return "Hi"; } // a normal non-virtual function

    virtual std::string_view getName() const { return "Base"; } // a normal virtual function

    virtual int getValue() const = 0; // a pure virtual function

    int doSomething() = 0; // Compile error: can not set non-virtual functions to 0
};
```

When we add a pure virtual function to our class, we are effectively saying, “it is up to the derived classes to implement this function”.
Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated! Consider what would happen if we could create an instance of Base:

21.16.2 Pure virtual functions with definitions

It turns out that we can create pure virtual functions that have definitions:

```cpp
#include <string>
#include <string_view>

class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name {};

public:
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

    const std::string& getName() { return m_name; }
    virtual std::string_view speak() const = 0; // The = 0 means this function is pure virtual

    virtual ~Animal() = default;
};

std::string_view Animal::speak() const  // even though it has a definition
{
    return "buzz";
}
```

In this case, speak() is still considered a pure virtual function because of the "= 0" (even though it has been given a definition) and Animal is still considered an abstract base class (and thus can't be instantiated). Any class that inherits from Animal needs to provide its own definition for speak() or it will also be considered an abstract base class.
When providing a definition for a pure virtual function, the definition must be provided separately (not inline).
This paradigm can be useful when you want your base class to provide a default implementation for a function, but still force any derived classes to provide their own implementation. However, if the derived class is happy with the default implementation provided by the base class, it can simply call the base class implementation directly.

21.16.3 Interface classes

An interface class is a class that has no member variables, and where all of the functions are pure virtual! Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.
Interface classes are often named beginning with an I. Here's a sample interface class:

```
#include <string_view>

class IErrorLog
{
public:
    virtual bool openLog(std::string_view filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(std::string_view errorMessage) = 0;

    virtual ~IErrorLog() {} // make a virtual destructor in case we delete
        an IErrorLog pointer, so the proper derived destructor is called
};
```

Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated.

21.17 Virtual base classes

https://www.learncpp.com/cpp-tutorial/virtual-base-classes/

21.18 Object slicing

https://www.learncpp.com/cpp-tutorial/object-slicing/

21.19 Dynamic casting

https://www.learncpp.com/cpp-tutorial/dynamic-casting/
In this lesson, we'll continue by examining another type of cast: dynamic_cast.

21.19.1 The need for `dynamic_cast`

When dealing with polymorphism, you'll often encounter cases where you have a pointer to a base class, but you want to access some information that exists only in a derived class.
Consider the following (slightly contrived) program:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Base
{
protected:
  int m_value{};

public:
  Base(int value)
    : m_value{value}
  {
  }

  virtual ~Base() = default;
};

class Derived : public Base
{
protected:
  std::string m_name{};

public:
  Derived(int value, std::string_view name)
    : Base{value}, m_name{name}
  {
  }

  const std::string& getName() const { return m_name; }
};

Base* getObject(bool returnDerived)
{
  if (returnDerived)
    return new Derived{1, "Apple"};
  else
    return new Base{2};
}

int main()
{
  Base* b{ getObject(true) };

  // how do we print the Derived object's name here, having only a Base
      pointer?

  delete b;

  return 0;
}
```

In this program, function getObject() always returns a Base pointer, but that pointer may be pointing to either a Base or a Derived object. In the case where the Base pointer is actually pointing to a Derived object, how would we call Derived::getName()?

One way would be to add a virtual function to Base called getName() (so we could call it with a Base pointer/reference, and have it dynamically resolve to Derived::getName()). But what would this function return if you called it with a Base pointer/reference that was actually pointing to a Base object? There isn't really any value that makes sense. Furthermore, we would be polluting our Base class with things that really should only be the concern of the Derived class.

We know that C++ will implicitly let you convert a Derived pointer into a Base pointer (in fact, getObject() does just that). This process is sometimes called **upcasting**. However, what if there was a way to convert a Base pointer back into a Derived pointer? Then we could call Derived::getName() directly using that pointer, and not have to worry about virtual function resolution at all.

21.19.2 dynamic_cast

C++ provides a casting operator named dynamic_cast that can be used for just this purpose. Although dynamic casts have a few different capabilities, by far the most common use for dynamic casting is for converting base-class pointers into derived-class pointers. This process is called downcasting.
Using dynamic_cast works just like static_cast. Here's our example main() from above, using a dynamic_cast to convert our Base pointer back into a Derived pointer:

```cpp
int main()
{
  Base* b{ getObject(true) };

  Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert
      Base pointer into Derived pointer

  std::cout << "The name of the Derived is: " << d->getName() << '\n';

  delete b;

  return 0;
}
```

This prints:

```
The name of the Derived is: Apple
```

21.19.3 dynamic_cast failure

The above example works because b is actually pointing to a Derived object, so converting b into a Derived pointer is successful.
However, we've made quite a dangerous assumption: that b is pointing to a Derived object. What if b wasn't pointing to a Derived object? This is easily tested by changing the argument to getObject() from true to false. In that case, getObject() will return a Base pointer to a Base object. When we try to dynamic_cast that to a Derived, it will fail, because the conversion can't be made.
If a dynamic_cast fails, the result of the conversion will be a null pointer.
Because we haven't checked for a null pointer result, we access d-¿getName(), which will try to dereference a null pointer, leading to undefined behavior (probably a crash).
In order to make this program safe, we need to ensure the result of the dynamic_cast actually succeeded:

```cpp
int main()
{
  Base* b{ getObject(true) };

  Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert
      Base pointer into Derived pointer

  if (d) // make sure d is non-null
    std::cout << "The name of the Derived is: " << d->getName() << '\n';

  delete b;

  return 0;
}
```

Rule: Always ensure your dynamic casts actually succeeded by checking for a null pointer result.

21.19.4 Downcasting with static_cast

It turns out that downcasting can also be done with static_cast. The main difference is that static_cast does no runtime type checking to ensure that what you're doing makes sense. This makes using static_cast faster, but more dangerous. If you cast a Base* to a Derived*, it will "succeed" even if the Base pointer isn't pointing to

a Derived object. This will result in undefined behavior when you try to access the resulting Derived pointer (that is actually pointing to a Base object).
If you're absolutely sure that the pointer you're downcasting will succeed, then using static_cast is acceptable.

`dynamic_cast` and references

`dynamic_cast` vs `static_cast`

New programmers are sometimes confused about when to use static_cast vs dynamic_cast. The answer is quite simple: use static_cast unless you're downcasting, in which case dynamic_cast is usually a better choice. However, you should also consider avoiding casting altogether and just use virtual functions.

Downcasting vs virtual functions

There are some developers who believe dynamic_cast is evil and indicative of a bad class design. Instead, these programmers say you should use virtual functions.
In general, using a virtual function should be preferred over downcasting. However, there are times when downcasting is the better choice:

- When you can not modify the base class to add a virtual function (e.g. because the base class is part of the standard library)
- When you need access to something that is derived-class specific (e.g. an access function that only exists in the derived class)
- When adding a virtual function to your base class doesn't make sense (e.g. there is no appropriate value for the base class to return). Using a pure virtual function may be an option here if you don't need to instantiate the base class.

Chapter 22

Debugging

Hello, here is some text without a meaning...