

Learncpp summary

**Summary: C++**

AgC86

-

Learncpp

**Author Note**

All of the work done is made possible thanks to the creators of the Learncpp website.

## Contents

1	Keywords.....	5
2	Introduction / Getting started .....	12
2.1	Introduction to programming languages .....	12
2.2	Introduction to C / C++ .....	17
2.3	Introduction to C++ development .....	21
2.4	Introduction to the compiler, linker and libraries .....	22
2.5	Configuring your compiler .....	25
3	Basics.....	29
3.1	Statements and the structure of a program .....	29
3.2	Comments .....	30
3.3	Introduction to objects and variables .....	32
3.4	Variable assignment.....	35
3.5	Variable initialization .....	36
3.6	Introduction to iostream.....	39
3.7	Uninitialized variables and undefined behavior .....	44
3.8	Keywords and naming identifiers .....	46
3.9	Whitespace and basic formatting .....	48
3.10	Introduction to literals .....	55
3.11	Introduction to Operators.....	56
3.12	Introduction to expressions .....	59
3.13	Chapter summary.....	61
4	Functions and files .....	63
4.1	Introduction to functions .....	63
4.2	Function return values (value-returning functions).....	67
4.3	Void functions (non-value returning functions).....	72
4.4	Introduction to function parameters and arguments.....	73
4.5	Introduction to local scope .....	77
4.6	Why functions are useful, and how to use them effectively .....	81
4.7	Forward declarations and definitions .....	82
4.8	Programs with multiple code files .....	87
4.9	Naming collisions and an introduction to namespaces .....	89
4.10	Introduction to the preprocessor .....	93
4.11	Header files .....	96
4.12	Header guards.....	100
4.13	How to design your first programs .....	104
4.14	Chapter 2 summary.....	105
5	Debugging .....	107
5.1	Syntax and semantic errors.....	107
5.2	Process .....	107
5.3	Strategy .....	107
6	Fundamental data types .....	108

6.1	Introduction to fundamental data types .....	108
6.2	Void .....	111
6.3	Object sizes and the sizeof operator.....	112
6.4	Signed integers.....	115
6.5	Unsigned integers, and why to avoid them .....	118
6.6	Fixed-width integers and size_t .....	120
6.7	Introduction to scientific notation .....	124
6.8	Floating point numbers .....	125
6.9	Boolean values .....	129
6.10	Boolean variables .....	129
6.11	Integer to Boolean conversion.....	129
6.12	Introduction to if statements.....	131
6.13	Chars .....	134
6.14	Introduction to type conversion and static_cast .....	138
6.15	Fundamental data types summary .....	141
7	Constants and Strings.....	143
7.1	Constant variables (named constants) .....	143
7.2	Literals.....	148
7.3	Numeral systems (decimal, binary, hexadecimal, and octal) .....	151
7.4	Constant expressions and compile-time optimization (run or compile-time).....	153
7.5	Constexpr variables (compile-time).....	161
7.6	The conditional operator .....	164
7.7	Inline functions and variables (may or may not inline) .....	166
7.8	Constexpr functions (run or compile-time) and Consteval functions (compile-time) ..	175
7.9	Introduction to std::string.....	185
7.10	Introduction to std::string_view .....	192
7.11	std::string_view (part 2).....	195
7.12	Chapter 5 summary and quiz .....	195



## 1 Keywords

**Computer program**, application

**Hardware**, physical computer parts

**Bit**, binary digit

**Assembler**, a program that translates assembly language into machine language

**Compiler**, a program (or collection of programs) that reads source code (typically written in a high-level language) and translates it into some other language (typically a low-level language, such as assembly or machine language, etc.).

**Interpreter**, a program that directly executes the instructions in the source code without requiring them to be compiled into an executable first.

**ANSI**, the American National Standards Institute

**ISO**, the International Organization for Standardization

**Language standard** or **language specification**, each new formal release of the language.

**Linker**, the linker's job is to combine all of the object files and produce the desired output file (typically an executable file). This process is called **linking**.

**Building**, the full process of converting source code files into an output file.

**Build configurations** or **build target**, a collection of project settings that determines how your IDE will build your project.

**Compiler extensions**, many compilers implement their own changes to the language, often to enhance compatibility with other versions of the language, or for historical reasons.

**Diagnostics**, if you have done something that definitively violates the rules of the language, the compiler is required to emit a diagnostic message.

**Statement**, a type of instruction that causes the program to perform some action.

**Identifier**, the name of a function (or object, type, template, etc.).

**Random Access Memory (RAM)**, the main memory in a computer

**Object**, a region of storage (usually memory) that can store a value, and has other associated properties. Typically refers to an unnamed object in memory, a variable, or a function.

**Variable**, an object that has been named (**identifier**).

**Definition**, a special kind of declaration statement for creating variables.

**Instantiation**, the object gets created and assigned a memory address.

**Instance**, instantiated object

**Data type (Type)**, determines what kind of value (e.g. a number, a letter, text, etc.) the object will store.

**Assignment**, the process of giving a value to a variable **after** it has been defined.

**Initialization**, giving an initial value to an object at the point of definition.

**Uniform or brace initialization**, list initialization

**Zero initialization**, default initialization

**Buffer**, a region of memory set aside to store a collection of requested data.

**Flushing**, transferring collected data from the buffer to its destination.

**Uninitialized**, an object that hasn't been given a known value.

**Undefined behavior (UB)**, the result of executing code whose behavior is not well-defined by the C++ language.

**Implementation-defined behavior**, the result of executing code whose behavior is defined by the compiler.

**Unspecified behavior**, this is almost identical to implementation-defined behavior. The only difference here is that the implementation (**compiler**) doesn't have to document why it behaves a certain way.

**Reserved words**, words reserved to the C++ language such as keywords.

**Operator Arity**, the number of operands that an operator takes as input.

**PEMDAS**, Parenthesis → Exponents → Multiplication & Division → Addition & Subtraction.

**Side effects**, an operator (or function) that has some observable effect beyond producing a return value.

**Metasyntactic variable** or **placeholder names**, a placeholder name for a function or variable when the name is unimportant to the demonstration of some concept.

**Status codes** or **exit codes**, the return value of a function is passed back to the caller of the function, the status code is passed back to the caller. This return value has a defined meaning (e.g. `EXIT_SUCCESS`).

**DRY**, don't repeat yourself

**WET**, write everything twice

**Modular programming**, the ability to write a function, test it, ensure that it works, and then know that we can reuse it as many times as we want and it will continue to work (so long as we don't modify the function -- at which point we'll have to retest it).

**Unreferenced parameters**, functions that have parameters that are not used in the body of the function.

**Lifetime**, an object's lifetime is defined to be the time between its creation and destruction. (**runtime** property)

**Scope**, an identifier's scope determines where the identifier can be seen and used within the source code. (**compile-time** property)

**Out of scope**, an identifier is out of scope anywhere it cannot be accessed within the code.

**Going out of scope**, we say an object goes out of scope at the end of the scope (the end curly brace) in which the object was instantiated.

**Temporary object** or **anonymous object**, an unnamed object that is created by the compiler to store a value temporarily.

**One task rule** or **separation of concerns (SoC)**, a design principle to separate an application into units, with minimal overlapping between the functions of the individual units.

**Function declaration statement** or **function prototype**, the function declaration consists of the function's return type, name, and parameter types, terminated with a semicolon. The names of the parameters can be optionally included. The function body is not included in the declaration.

**Pure declaration**, In C++, all definitions are declarations. Conversely, not all declarations are definitions. Declarations that aren't definitions are called pure declarations.

**One definition rule (ODR)**, Only one definition of any variable, function, class type, enumeration type, concept(since C++20) or template is allowed in any one translation unit (some of these may have multiple declarations, but only one definition is allowed).

**Qualified name**, an identifier that includes a namespace prefix.

**Using-directive statement**, allows us to access the names in a namespace without using a namespace prefix.

**Preprocessor directive or directives**, instructions that start with a # symbol and end with a newline (NOT a semicolon).

**Translation unit**, when the preprocessor has finished processing a code file, the result is called a translation unit.

**Translation**, the entire process of **preprocessing**, **compiling**, and **linking**.

**Transitive includes**, when your code file #includes the first header file, you'll also get any other header files that the first header file includes (and any header files those include, and so on). These additional header files are sometimes called transitive includes, as they're included implicitly rather than explicitly.

**Header guard or include guard**, conditional compilation directives that ensure proper transitive includes, avoiding **ODR** violations.

**Call stack**, a list of all the active functions that have been called to get to the current point of execution. The call stack includes an entry for each function called, as well as which line of code will be returned to when the function returns. Whenever a new function is called, that function is added to the top of the call stack. When the current function returns to the caller, it is removed from the top of the call stack, and control returns to the function just below it.

**Fundamental data types or primitive types**, data types that are built into and supported by C++ (void, std::nullptr\_t, integral types, floating-point types).

**IEEE**, Institute of Electrical and Electronics Engineers

**Sign**, the attribute of being positive, negative, or zero is called the number's sign.

**Sign bit**, a single bit used to store the sign of the number.



**Magnitude bits**, the non-sign bits that determine the magnitude of the number.

**Shorthand types**, types that do not use a (int) suffix or signed prefix.

**Fixed-width integers**, int types that have a fixed-width across CPU architectures. C99 defined them in the **stdint.h**. Unfortunately, they may not be defined on all architectures and may be slower in terms of processing speed.

**Fast integers**, fast types defined by C++ to address the issue of processing speed with fixed-width integers defined in C99. These can be found in the **cstdint** header along with the **stdint.h** types.

**Least integers**, least types defined by C++ to address the issue of architecture support with fixed-width integers defined in C99. They give you the **smallest** (least) integer type.

**std::size\_t**, an alias for an implementation-defined unsigned integral type.

**Significant digits** or **significant figures**, refers to scientific notation of the digits in the significand (the part before the 'e'). The more significant digits, the more precise a number is.

**Condition** or **conditional expression**, an expression that evaluates to a Boolean value.

**Early return**, a return statement that is not the last statement in a function.

**Integral type**, any type where the underlying value is stored as an integer. Obviously types like int, short, etc. but less obvious types like bool (0 or 1), chars (ASCII values).

**ASCII**, American Standard Code for Information Interchange

**Type qualifier** or **qualifier**, a keyword that is applied to a type that modifies how that type behaves. As of C++23, C++ only has two type qualifiers: **const** and **volatile**.

**Cv-qualifier**, in technical documentation, the const and volatile qualifiers are often referred to as cv-qualifiers.

**As-if rule**, it says that 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**".

**Constant variable** or **named constants**, variables with fixed values that cannot be changed.

**Constant expression**, expression that contains only compile-time constants and / or operators / functions that support **compile-time evaluation**.

**Runtime expression**, expression that is not a constant expression.

**Compile-time constant**, a constant whose value must be known at **compile time**.

**Runtime constant**, Const variables that are not compile-time constants. Runtime constants **cannot be used in a constant expression**.

**Immediate functions**, a function that must evaluate at **compile-time**, otherwise a compile error will result.

**Read-only**, we can access and use the value being viewed, but we cannot modify it.



## C++

### 2 Introduction / Getting started

#### 2.1 Introduction to programming languages

A **computer program** (also commonly called an **application**) is a **set of instructions** that the computer can perform in order to perform some task(s). The process of creating a program is called programming. Programmers typically create programs by producing source code (commonly shortened to code), which is a list of commands typed into one or more text files.

The collection of **physical** computer **parts** that make up a computer and execute programs is called the **hardware**. When a computer program is loaded into memory and the hardware sequentially executes each instruction, this is called **running** or **executing** the program.

##### 2.1.1 Machine language

The collection of physical computer parts that make up a computer and execute programs is called the **hardware**. When a computer program is loaded into memory and the hardware sequentially executes each instruction, this is called **running** or **executing** the program.

Example: **1011000 0110001**

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 that make up a single command varies.

Example:

- Some CPUs process instructions are always 32 bits long
- Other CPUs from the x86/x64 family have instructions that can be variable in length

Each set of binary digits is interpreted by the CPU into a command to do a very specific job, such as **compare these two numbers**, or **put this number in that memory location**.

However, because different **CPUs have different instruction sets**, instructions that were written for one CPU type could not be used on a CPU that didn't share the same instruction set. Back in the day this meant that programs generally weren't **portable**.

### 2.1.2 Assembly language

Because machine language is so hard for humans to read and understand, assembly language was invented. In an assembly language, each instruction is identified by a short abbreviation (**rather than a set of bits**), and names and other numbers can be used.

Here is the same instruction as above in assembly language: **mov al, 061h**

This makes assembly much easier to read and write than machine language. However, the **CPU cannot understand assembly language directly**. Instead, the assembly **program must be translated** into machine language before it can be executed by the computer. This is done by using a program called an **assembler**.

Assembly languages still require a lot of instructions to do even simple tasks. While the individual instructions themselves are somewhat human readable, understanding what an entire program is doing can be challenging. Assembly isn't portable, a program written in assembly for one CPU will likely not work on hardware that uses a different instruction set, and would have to be rewritten or extensively modified.

### 2.1.3 High-level languages

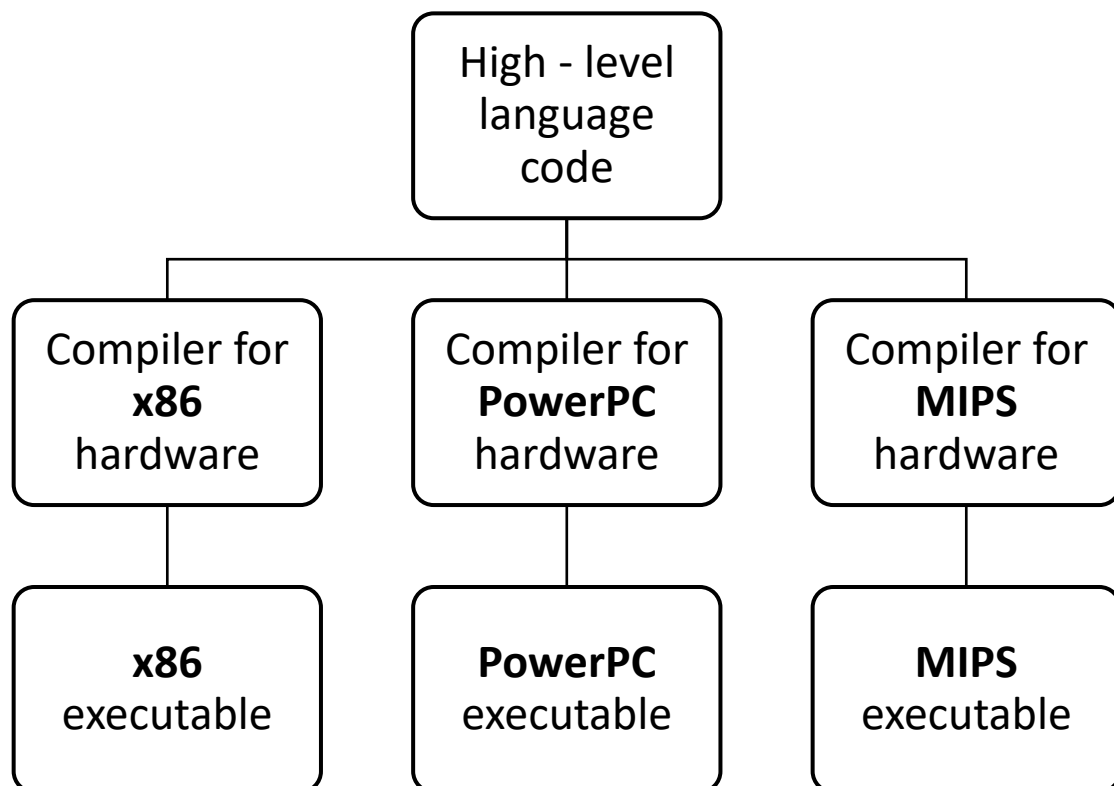
Now that we know some of the downsides:

- Hard to read
- Not portable
- A lot of work for simple instructions

This is where higher-level languages come into play. To address the readability and portability concerns, new programming languages such as C, C++, Pascal (and later, languages such as Java, Javascript, and Perl) were developed. These languages are called **high level languages**, as they are designed to allow the programmer to write programs without having to be as concerned about what kind of computer the program will be run on.

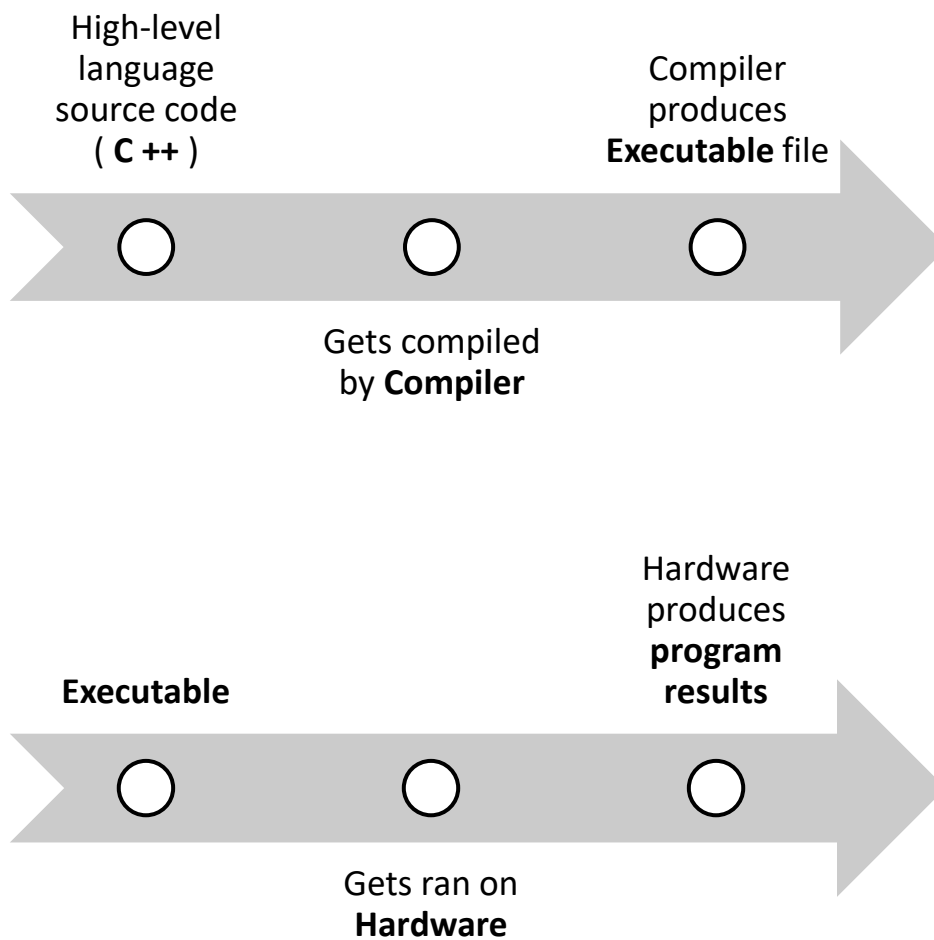
Here is the same instruction as above in C/C++: **a = 97;**

Much like assembly programs, programs written in high level languages must be translated into a format the computer can understand before they can be run. There are two primary ways this is done: compiling and interpreting. Most languages can be compiled or interpreted. Traditionally 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**. Programs can be compiled (or interpreted) for many different systems, **and you don't have to change the program to run on different CPUs** (you just recompile for that CPU).



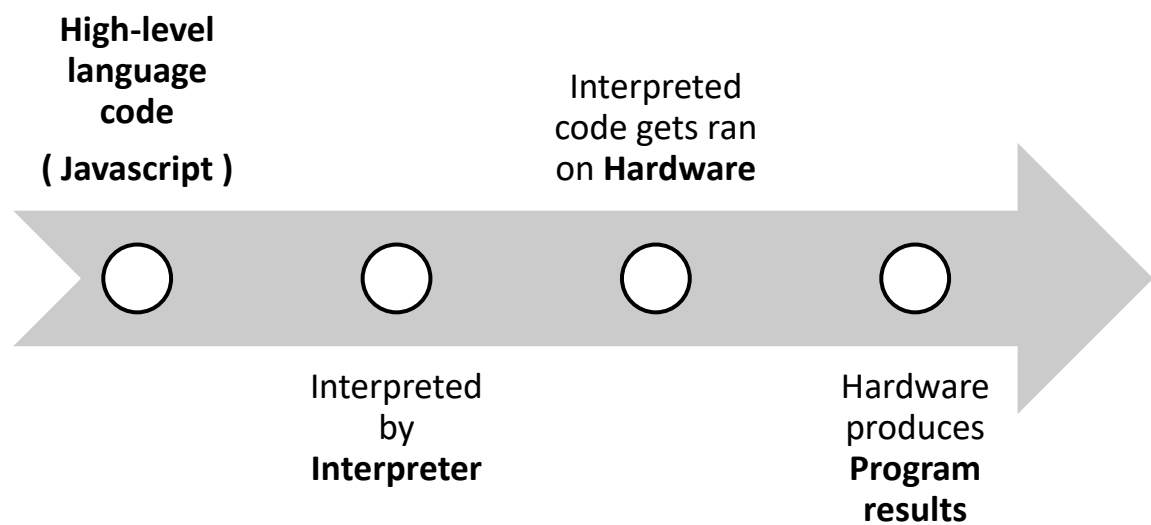
### 2.1.3.1 Compiling

A **compiler** is a program (or collection of programs) that reads source code (typically written in a high-level language) and **translates it into** some other language (typically a **low-level language**, such as assembly or machine language, etc.). Most often, these low-level language files are then combined into an executable file (containing machine language instructions) that can be run or distributed to others. Notably, **running** the executable file **does not require** the **compiler** to be installed (because the **code has already been compiled and put into the executable file**).



### 2.1.3.2 Interpreter

An **interpreter** is a program that **directly executes** the **instructions** in the source code without requiring them to be compiled into an executable 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.**





## 2.2 Introduction to C / C++

### 2.2.1 *Before C++, there was C*

C was created by **Dennis Ritchie** in 1972, primarily to **serve as a systems programming language** (To write Operating Systems with).

Goals for C:

- Minimalistic
- Easy to compile
- Efficient access to memory
- Efficient code
- Self-contained (Not reliant on other languages)

#### 2.2.1.1 Brief history: C

Essentially, it was designed to give the programmer a lot of control, while still encouraging hardware and operating systems independence.

##### 2.2.1.1.1 1973: *Unix*

It was so flexible that in 1973 the **Unix** operating system was written in C and Assembly.

##### 2.2.1.1.2 1978: *“The C Programming Language”*

In 1978, Brian **Kernighan** and Dennis **Ritchie** published a book called **“The C Programming Language”**. This book, which was commonly known as **K&R**, provided an informal specification for the language and **became a de facto standard**.

**2.2.1.1.3 1983: Forming of ANSI committee**

In 1983, the American National Standards Institute (ANSI) formed a committee to establish a formal standard for C.

**2.2.1.1.4 1989: Release: C89 standard / ANSI C**

In 1989 (committees take forever to do anything), they finished, and released the C89 standard, more commonly known as ANSI C.

**2.2.1.1.5 1990: Release: C90 standard: An adoption of ANSI C by the ISO**

In 1990 the International Organization for Standardization (ISO) adopted ANSI C (with a few minor modifications). This version of C became known as C90. Compilers eventually became ANSI C/C90 compliant, and programs desiring maximum portability were coded to this standard.

**2.2.1.1.6 1999: Release: C99 standard: An extended version of C90**

In 1999, the ISO committee released a new version of C called C99. C99 adopted many features which had already made their way into compilers as extensions, or had been implemented in C++.

**2.2.1.1.7 Later releases**

- 2011: **C11**
- 2017: **C17**
- 2023: **C23**

### 2.2.1.2 Brief history: C++

C++ was created by **Bjarne Stroustrup** in 1979, primarily to **serve as an extension to C**. It is best thought of as a “superset” of C. Though new releases of C like the **C99 standard** introduced new features that weren’t in C++.

Goals for C:

- More features
- Object-oriented

#### 2.2.1.2.1 1998: C++ standardized by the ISO Committee

C++ was standardized in 1998 by the ISO committee (this means the ISO standards committee approved a document describing the C++ language, to help ensure all compilers adhere to the same set of standards)

#### 2.2.1.2.2 2003: Release: C++03 standard

A minor update to the language was released in 2003 (called C++03).

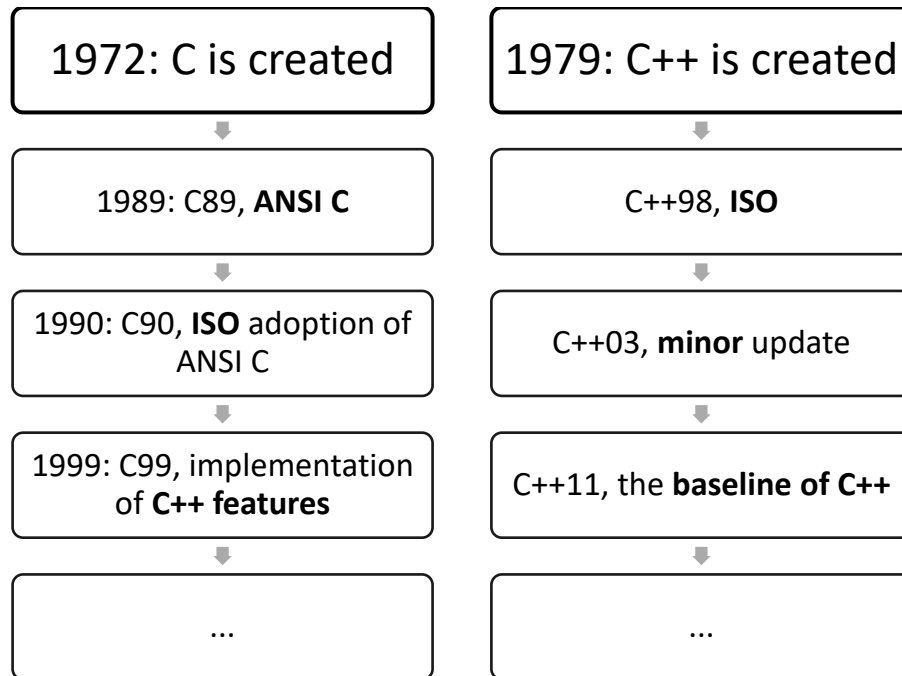
#### 2.2.1.2.3 Later releases

Five major updates to the C++ language (**C++11, C++14, C++17, C++20, and C++23**) have been made since then, each adding additional functionality. **C++11 in particular added a huge number of new capabilities, and is widely considered to be the new baseline version of the language.** Future upgrades to the language are expected every three or so years.

Each new formal release of the language is called a **language standard** (or **language specification**).

Standards are named after the year they are released in. For example, there is no C++15, because there was no new standard in 2015.

### 2.2.1.3 Summary of language standards

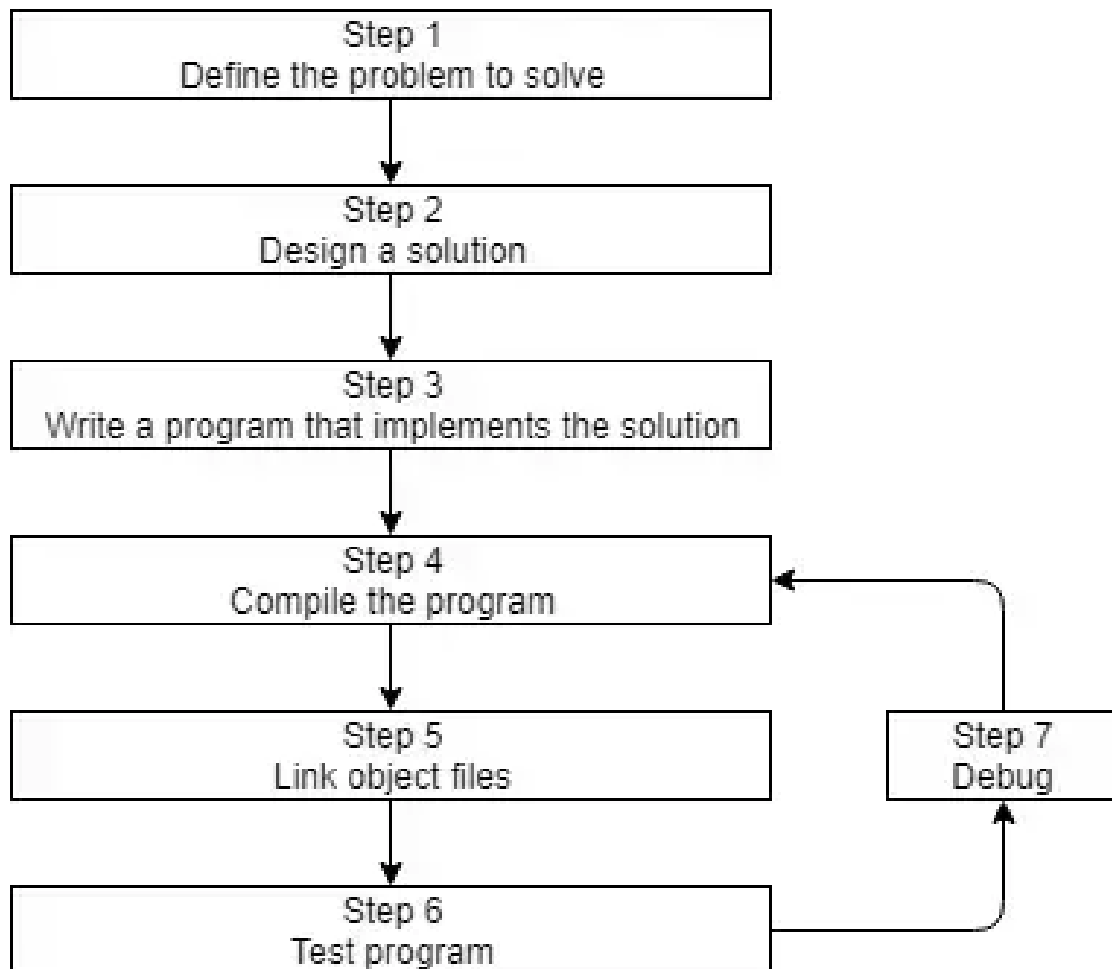


### 2.2.2 C / C++ philosophy

The underlying design philosophy of C and C++ can be summed up as “**trust the programmer**” -- which is **both wonderful and dangerous**. C++ is designed to allow the programmer a **high degree of freedom** to do what they want. However, **this also means the language often won’t stop you from doing things that don’t make sense**, because it will assume you’re doing so for some reason it doesn’t understand. There are quite a few pitfalls that new programmers are likely to fall into if caught unaware. **This is one of the primary reasons why knowing what you shouldn’t do in C/C++ is almost as important as knowing what you should do.**

## 2.3 Introduction to C++ development

### 2.3.1 C++ development flow



## 2.4 Introduction to the compiler, linker and libraries

### 2.4.1 Compiling your source code

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. The compiler does two important tasks:

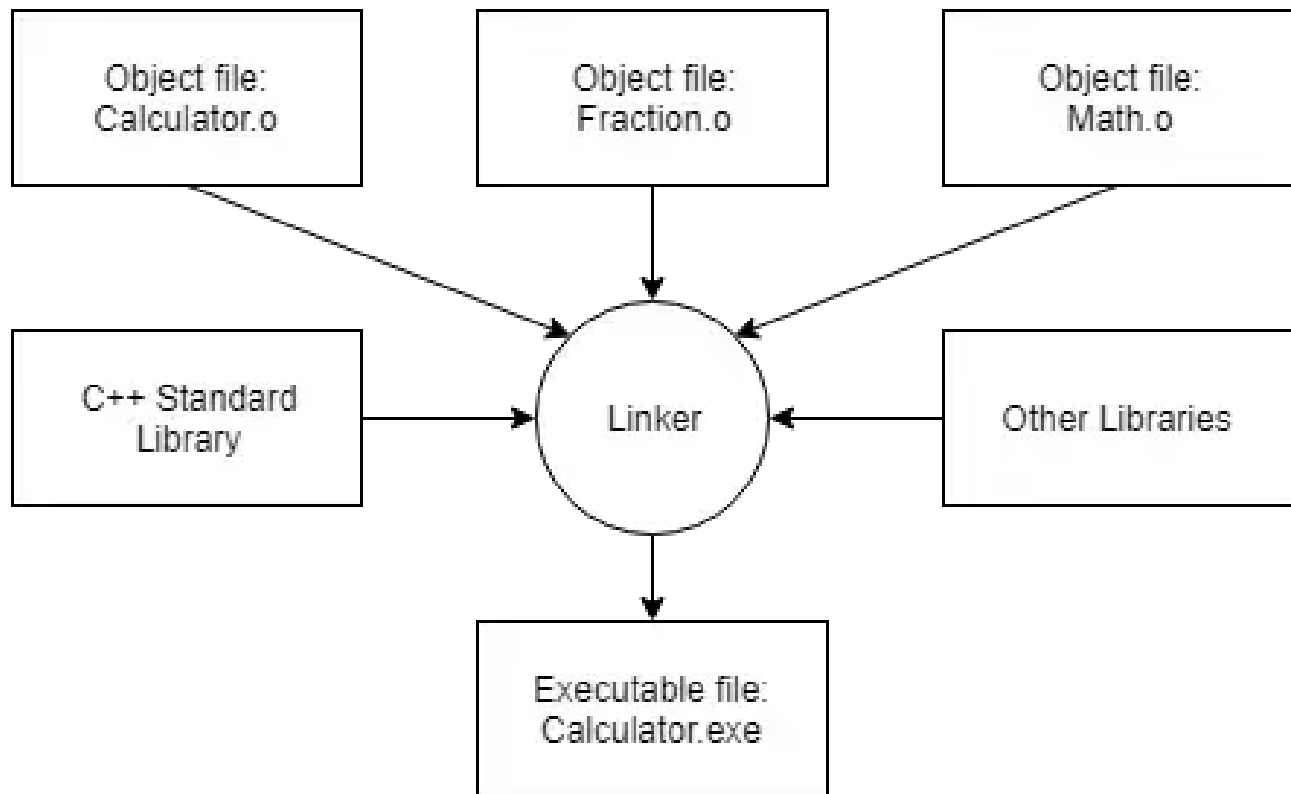
- **Checks** your C++ **code** to make sure it follows the rules of the C++ language.
  - **If it does not, the compiler will give you an error** (and the corresponding line number) to help pinpoint what needs fixing. **The compilation process will also be aborted until the error is fixed.**
- **Translates** your C++ **code** into machine language instructions.
  - These instructions are **stored in an intermediate** file called an **object file**. The **object file also contains metadata that is required or useful in subsequent steps**. These files carry the **.o / .obj extension** and will have the **same name as your .cpp Files**.

### 2.4.2 Linking object files and libraries

**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** (typically an executable file). This process is called **linking**.

The linker's flow:

- Read each object file to make sure they are valid
- Resolve cross-file dependencies
  - These are things you use something from one file to the other
  - **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**.
- Linking library files
  - A **library file** is a collection of **precompiled code** that has been "packaged up" for reuse in other programs.
  - C++ comes with an extensive library called the **C++ Standard Library** (usually shortened to *standard library*) that provides a set of useful capabilities for use in your programs.
  - You can also optionally **link other libraries**.
    - Example: C++ doesn't let you play sounds by standard. And figuring out how to code sound is a lot of work. Instead, you'd probably download a library that already knew how to do those things, and use that.



### 2.4.3 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**.

#### 2.4.3.1 Build types

- **Build**
  - **compiles** all *modified* code files in the project or workspace/solution, and **then links the object files into an executable**.
  - If no code files have been modified since the last build, this option does nothing.
- **Clean**
  - **removes all cached objects and executables** so the next time the project is built, all files will be recompiled and a new executable produced.
- **Rebuild**
  - does a “clean”, followed by a “build”.
- **Compile**
  - recompiles a **single code file** (regardless of whether it has been cached previously).
  - This option **does not invoke the linker** or produce an executable.
- **Run/start**
  - **executes the executable from a prior build**.
  - Some IDEs (e.g. Visual Studio) will **invoke a “build” before doing a “run”** to ensure you are running the latest version of your code.

For this summary I’m not going to go into IDE’s to use or how to build your first program or what IDE’s look like.



## 2.5 Configuring your compiler

### 2.5.1 *Build configurations*

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.

#### 2.5.1.1 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.

#### 2.5.1.2 Release configuration

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).

### 2.5.2 *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**.

In visual studio this setting is called the **Conformance mode** and **belongs to the C/C++ language settings**. It should be set to Yes or Permissive- (non-permissive).

### 2.5.3 *Warnings and error levels*

When you write your programs, the compiler will check to ensure you've followed the rules of the C++ language (assuming you've turned off compiler extensions).

#### 2.5.3.1 **Diagnostics**

If you have done something that definitively violates the rules of the language, the compiler is required to emit a **diagnostic message** (often called a **diagnostic** for short). The C++ standard does not define how diagnostic messages should be categorized or worded. However, there are some common conventions that compilers have adopted.

### 2.5.3.2 Errors

If compilation cannot continue due to the violation, then the compiler will emit an **error**, providing both line number containing the error, and some text about what was expected vs what was found. Errors stop the compilation from proceeding. **The actual error may be on that line, or on a preceding line.** Once you've identified and fixed the erroneous line(s) of code, you can try compiling again.

### 2.5.3.3 Warnings

If compilation can continue despite the violation, the compiler may decide to emit either an error or a **warning**. Warnings are similar to errors, but they do not halt compilation.

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.

**Best practice: Don't let warnings pile up, resolve them immediately (as if they were errors).**

**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 for things it finds strange.

**Best practice: Turn your warning levels up to the maximum, especially when learning.**

### 2.5.3.4 Treating warnings as errors

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).**

**Best practice: Enable "Treat warnings as errors". This will force you to resolve all issues causing warnings.**

### 2.5.4 *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.

#### 2.5.4.1 Code names for in-progress language standards

**Finalized language standards are named after the years in which they are finalized** (e.g. C++17 was finalized in 2017).

However, **when a new language standard is being worked on**, it's not clear in what year the finalization will take place. **Consequently, upcoming language standards are given code names**, which are then replaced by the actual names upon finalization of the standard. You may still see the code names used in places (particularly for the latest in-progress language standard that doesn't have a finalized name yet).

#### 2.5.4.2 Which language standard should you use?

**In professional environments, it's common to choose a language standard that is one or two versions back** from the latest standard (e.g. if C++20 were the latest 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.

**For personal projects and while learning, we recommend choosing the latest finalized standard, as there is little downside to doing so.**

All the information regarding the C++ standard can be found on: <https://isocpp.org/>.

**Even after a language standard is finalized, compilers supporting that language standard often still have missing, partial, or buggy support for certain features.**

### 3 Basics

#### 3.1 Statements and the structure of a program

##### 3.1.1 Statements

A computer program is a **sequence** of instructions that tell the computer what to do.

A **statement** is a type of instruction that causes the program to *perform some action*.

Statements are by far the most common type of instruction in a C++ program. This is because they are the **smallest independent unit of computation** in the C++ language. In that regard, they act much like sentences do in natural language. When we want to convey an idea to another person, we typically write or speak in sentences (not in random words or syllables). In C++, when we want to have our program do something, we typically write statements.

**Most (but not all) statements in C++ end in a semicolon.** If you see a line that ends in a semicolon, it's probably a statement.

**In a high-level language such as C++, a single statement may compile into many machine language instructions.**

##### 3.1.1.1 Different kinds of statements

- Declaration
- Jump
- Expression
- Compound
- Selection (Conditionals)
- Iteration (Loops)
- Try blocks

### 3.1.2 *Functions and the main function*

In C++, **statements are typically grouped into units called functions**. A function is a collection of statements that get executed sequentially (in order, from top to bottom). As you learn to write your own programs, you'll be able to create your own functions and mix and match statements in any way you please (we'll show how in a future lesson).

#### **Important rule:**

Every C++ program must have a **special function named main (all lower-case letters)**. When the program is run, the statements inside of main are **executed in sequential order**. Programs typically terminate (finish running) after the last statement inside function main has been executed (though programs may abort early in some circumstances, or do some cleanup afterwards).

### 3.1.3 *Identifiers*

In programming, the name of a **function (or object, type, template, etc.)** is called its **identifier**.

### 3.1.4 *Syntax and syntax errors*

In English, sentences are constructed according to specific grammatical rules that you probably learned in English class in school. For example, normal sentences end in a period. **The rules that govern how sentences are constructed in a language is called syntax**. If you forget the period and run two sentences together, this is a violation of the English language syntax.

**C++ has a syntax too: rules about how your programs must be constructed in order to be considered valid**. When you compile your program, the compiler is responsible for making sure your program follows the basic syntax of the C++ language. If you violate a rule, the compiler will complain when you try to compile your program, and issue you a syntax error.

## 3.2 **Comments**

A comment is like leaving a note in your source code. It is ignored by the compiler and is mostly to help programmers document the code in some way.

### **3.2.1 *Single-line comments***

We start by writing the `//` symbol. Everything before that gets compiled but everything after the symbol up until the new line is a comment.

### **3.2.2 *Multi-line comments***

We start by writing the `/*` symbol and end with the `*/` symbol. Everything in between the first and the last symbol gets ignored by the compiler. The use here is to have a “section” where you can write comments.

### **3.2.3 *Best practice***

Comment your code liberally, and write your comments as if speaking to someone who has no idea what the code does. Don't assume you'll remember why you made specific choices.

### 3.3 Introduction to objects and variables

You learned that the **majority of instructions in a program are statements**, and that **functions are groups of statements** that execute sequentially.

#### 3.3.1 Key Insight

**Programs are collections of instructions that manipulate data** to produce a desired result.

#### 3.3.2 Random Access Memory (RAM)

The main memory in a computer is called **Random Access Memory** (often called **RAM** for short). When we run a program, the operating system loads the program into RAM. Any **data that is hardcoded** into the program itself (e.g. text such as “Hello, world!”) is **loaded at this point**.

The operating system **also reserves some additional RAM for the program to use** while it is running. Common uses for this memory are to store values entered by the user, to store data read in from a file or network, or to store values calculated while the program is running (e.g. the sum of two values) so they can be used again later.

You can think of RAM as a series of numbered boxes that can be used to store data while the program is running.

**In some older programming languages (like Applesoft BASIC), you could directly access these boxes (e.g. you could write a statement to “go get the value stored in mailbox number 7532”).**

##### 3.3.2.1 Quick summary

- Executable get ran
  - OS directly loads hardcoded values into memory
  - OS reserves additional space for data to be manipulated while the program is running



### 3.3.3 Objects & Variables

In C++, **direct memory access is discouraged**. Instead, we **access memory indirectly through an object**. An object is a **region of storage** (usually memory) that can store a value, and has other associated properties (that we'll cover in future lessons). How the compiler and operating system work to assign memory to objects is beyond the scope of this lesson. But the key point here is that rather than say "go get the value stored in mailbox number 7532", we can say, "go get the value stored by this object". **This means we can focus on using objects to store and retrieve values, and not have to worry about where in memory those objects are actually being placed.**

#### 3.3.3.1 Key insight

An **object** is used **to store a value in memory**. A **variable is an object** that has been named (identifier). Naming our objects let us refer to them again later in the program.

#### 3.3.3.2 Nomenclature

In general programming, the term *object* typically refers to an unnamed object in memory, a variable, or a function. **In C++, the term *object* has a narrower definition that excludes functions.**

#### 3.3.3.3 Instantiation & Instances, a runtime occurrence

In order to create a variable, we use a special kind of declaration statement called a **definition**.

At compile time, when the compiler sees this statement, **it makes a note to itself** that we are defining a variable, giving it the name *x*, and that it is of type `int` (more on types in a moment). **From that point forward** (with some limitations that we'll talk about in a future lesson), **whenever the compiler sees the identifier *x*, it will know that we're referencing this variable.**

When the program is run (called **runtime**), the variable will be instantiated. **Instantiation** is a fancy word that **means the object will be created and assigned a memory address**. Variables must be instantiated before they can be used to store values. For the sake of example, **let's say that variable *x* is instantiated at memory location 140. Whenever the program uses variable *x*, it will access the value in memory location 140.** An instantiated object is sometimes called an **instance**.

### 3.3.4 Data types

A **data type** (more commonly just called a **type**) determines what kind of value (e.g. a number, a letter, text, etc.) the object will store. In C++, the type of a variable must be known at **compile-time** (when the program is compiled), and that type cannot be changed without recompiling the program. This means an integer variable can only hold integer values. If you want to store some other kind of value, you'll need to use a different type. Or do type conversion, which is something we'll get into more depth in, later on.

#### 3.3.4.1 Defining a variable of a certain type

Int a;	← This is a <b>definition</b>
Int a, b;	← This defines two variables
Int a, int b;	← Wrong, compiler error
Int a, double b;	← Wrong, compiler error

In both cases these variables are of the integer type which allows them to store a numeric value into memory (not floating-point numeric values). There is **no need to specify the type twice** when defining variables, since it's in the same statement the compiler will know what type, it is.

**Best practice:** Although the language allows you to do so, **avoid defining multiple variables of the same type in a single statement**. Instead, define each variable in a separate statement on its own line (and then **use a single-line comment to document what it is used for**).

### 3.4 Variable assignment

#### 3.4.1 Quick recap

In previous lessons we covered:

- **Instantiation** (the process of an object getting created and assigned to its memory address by the runtime system).
- **Definition** (Defining a variable by assigning it a type and identifier).

#### 3.4.2 Variable assignment

After a variable has been defined, you can give it a value (in a separate statement) using the = *operator*. This process is called **assignment**, and the = *operator* is called the **assignment operator**.

Float pi;	← Definition
pi = 3.14f;	← Assignment

By default, assignment copies the value on the right-hand side of the = *operator* to the variable on the left-hand side of the operator. This is called **copy assignment**. This can be used to manipulate our variable.

Example: Since the type is already defined and the compiler knows about it we can access the variable by its **identifier (name)** and copy a different value into it.

pi = 3.141592f;	← Not definition but assigned a different value
-----------------	---

### 3.5 Variable initialization

One downside of **assignment** is that it **requires at least two statements**: one to define the variable, and another to assign the value. **These two steps can be combined**. When an object is defined, you can optionally give it an initial value. **The process of specifying an initial value for an object is called initialization**, and the syntax used to initialize an object is called an **initializer**.

Float pi;	← Definition
pi = 3.14f;	← Assignment
Float pi = 3.14f;	← Defined & Assigned → Initialized

#### 3.5.1 Forms of initialization

```
int a;    // no initializer, just a definition (default initialization by compiler)

int b = 5; // initial value after equals sign (copy initialization)

int c ( 6 ); // initial value in parenthesis (direct initialization)

// List initialization methods (C++11) (preferred)

int d { 7 }; // initial value in braces (direct list initialization)

int e = { 8 }; // initial value in braces after equals sign (copy list initialization)

int f {}; // initializer is empty braces (value initialization)
```

### 3.5.1.1 Default initialization

When no initializer is provided (such as for variable `a` above), this is called **default initialization**. In most cases, default initialization performs no initialization, and **leaves a variable with an indeterminate value**.

### 3.5.1.2 Copy initialization

When an initial value is provided after an equals sign, this is called **copy initialization**. This form of initialization was inherited from C. Much like copy assignment, this copies the value on the right-hand side of the equals into the variable being created on the left-hand side.

**Copy initialization had fallen out of favor in modern C++ due to being less efficient than other forms of initialization for some complex types.** However, C++17 remedied the bulk of these issues, and copy initialization is now finding new advocates. **You will also find it used in older code (especially code ported from C), or by developers who simply think it looks more natural and is easier to read.**

### 3.5.1.3 Direct initialization

When an initial value is provided inside parenthesis, this is called **direct initialization**.

Direct initialization was initially introduced to allow for more efficient initialization of complex objects (those with class types, which we'll cover in a future chapter). **Just like copy initialization, direct initialization had fallen out of favor in modern C++**, largely due to being superseded by list initialization. However, we now know that list initialization has a few quirks of its own, and so direct initialization is once again finding use in certain cases.

#### 3.5.1.4 List initialization

The modern way to initialize objects in C++ is to use a form of initialization that makes use of curly braces. This is called **list initialization** (or **uniform initialization** or **brace initialization**). List initialization has an added benefit: “narrowing conversions” in list initialization are ill-formed. **This means that if you try to brace initialize a variable using a value that the variable cannot safely hold, the compiler is required to produce a diagnostic (usually an error).** When a variable is initialized using empty braces, value initialization takes place. **In most cases, value initialization will initialize the variable to zero (or empty, if that’s more appropriate for a given type).** In such cases where zeroing occurs, this is called **zero initialization**.

Essentially writing:

<code>int a {};</code> is the same as writing <code>int a = 0;</code>
---

#### 3.5.1.5 Best practice

**Prefer direct list initialization** (or value initialization) for initializing your variables.

- **Compiler errors** (a good way to identify if the type is compatible with the value you are trying to assign to it)
- **Zero initialization > Default initialization**

Prepend the `[[maybe_unused]]` attribute if you initialize a variable but **don’t want the compiler warning you that it’s not being used.**

### 3.6 Introduction to iostream

#### 3.6.1 *The input/output library*

The input/output library (io library) is **part of the C++ standard library** that deals with basic input and output. We'll use the functionality in this library to get input from the keyboard and output data to the console. **The io part of iostream stands for input/output.**

**To use the functionality defined within the iostream library, we need to include the iostream header at the top of any code file** that uses the content defined in iostream.

```
#include <iostream>
```

```
// rest of code that uses iostream functionality here
```

### 3.6.2 *Cout: Character output*

The *iostream* library contains a few predefined variables for us to use. One of the most useful is **std::cout**, which allows us to send data to the console to be printed as text. *cout* stands for “character output”.

```
#include <iostream> // for std::cout

int main()

{

    std::cout << "Hello world!"; // print Hello world! to console

    return 0;

}
```

In this program, we have included *iostream* so that we have access to *std::cout*. Inside our *main* function, we use *std::cout*, along with the **insertion operator (<<)**, to send the text *Hello world!* to the console to be printed. To print more than one thing on the same line, the insertion operator (<<) can be used multiple times in a single statement to concatenate (link together) multiple pieces of output.

```
int X {};
```

```
std::cout << "The variable named X is equal to: " << X;
```



### 3.6.2.1 Cout Buffer

Consider a rollercoaster ride at your favorite amusement park. Passengers show up (at some variable rate) and get in line. Periodically, a train arrives and boards passengers (up to the maximum capacity of the train). When the train is full, or when enough time has passed, the train departs with a batch of passengers, and the ride commences. Any passengers unable to board the current train wait for the next one.

This analogy is similar to how output sent to `std::cout` is typically processed in C++. Statements in our program request that output be sent to the console. However, that output is typically not sent to the console immediately. Instead, the requested output “gets in line”, and is stored in a region of memory set aside to collect such requests (called a **buffer**). Periodically, the buffer is **flushed**, meaning all of the data collected in the buffer is transferred to its destination (in this case, the console).

### 3.6.3 Endl: End line

So, what happens when we write the following?

```
std::cout << "Hi!";  
  
std::cout << "My name is Alex.";
```

Result: Hi!My name is Alex. ← Notice there is **no space between sentence 1 and 2**, there is **also no newline**. It all gets written on the same line.

If we want to print separate lines of output to the console, we need to tell the console to move the cursor to the next line. We can do that by outputting a newline. A **newline** is an **OS-specific** character or sequence of characters that moves the cursor to the start of the next line.

```
std::cout << "Hi!" << std::endl; // std::endl will cause the cursor to move to the next line  
  
std::cout << "My name is Alex." << std::endl;  
  
Hi!  
  
My name is Alex.
```

### 3.6.3.1 Endl vs. \n

Using `std::endl` is often inefficient, as it actually does **two jobs**: it outputs a **newline** (moving the cursor to the next line of the console), **and** it **flushes the buffer** (which is slow). If we output multiple lines of text ending with `std::endl`, we will get multiple flushes, which is slow and probably unnecessary.

When outputting text to the console, we typically don't need to explicitly flush the buffer ourselves. **C++'s output system is designed to self-flush periodically**, and it's both simpler and more efficient to let it flush itself.

**To output a newline without flushing the output buffer, we use `\n` (inside either single or double quotes), which is a special symbol that the compiler interprets as a newline character.** `\n` moves the cursor to the next line of the console without causing a flush, so it will typically perform better. `\n` is also more concise to type and can be embedded into existing double-quoted text.

### 3.6.3.2 Best practice

- Output a newline **whenever a line of output is complete**.
- Prefer `\n` over `std::endl`

### 3.6.4 Cin: Character input

`std::cin` is another predefined variable that is defined in the `iostream` library.

Whereas `std::cout` prints data to the console using the insertion operator (`<<`), `std::cin` (which stands for "character input") reads input from keyboard using the **extraction operator** (`>>`). The input must be stored in a variable to be used.

```
int x{};           // define variable x to hold user input (and value-initialize it)

std::cin >> x;     // get number from keyboard and store it in variable x

std::cout << "You entered " << x << '\n';
```

#### 3.6.4.1 Best practice

There's some debate over whether it's necessary to initialize a variable immediately before you give it a user provided value via another source (e.g. `std::cin`), since the user-provided value will just overwrite the initialization value. In line with our previous recommendation that variables should always be initialized, **best practice is to initialize the variable first.**

### 3.7 Uninitialized variables and undefined behavior

Unlike some programming languages, C/C++ does not automatically initialize most variables to a given value (such as zero). When a variable that is not initialized is given a memory address to use to store data, **the default value of that variable is whatever (garbage) value happens to already be in that memory address!** A variable that has not been given a known value (through initialization or assignment) is called an **uninitialized variable**.

#### 3.7.1 Nomenclature

Many readers expect the terms “initialized” and “uninitialized” to be strict opposites, but they really aren’t! In common language, “initialized” means the object was provided with an initial value at the point of definition. “Uninitialized” means the object has not been given a known value yet (through any means, including assignment). Therefore, an object that is not initialized but is then assigned a value is no longer uninitialized (because it has been given a known value).

To recap:

- **Initialized** = The object is given a known value at the point of definition.
- **Assignment** = The object is given a known value beyond the point of definition.
- **Uninitialized** = The object has not been given a known value yet.

#### 3.7.2 As an aside

This lack of initialization is a performance optimization inherited from C, back when computers were slow. Imagine a case where you were going to read in 100,000 values from a file. **In such case, you might create 100,000 variables, then fill them with data from the file.**

If C++ initialized all of those variables with default values upon creation, this would result in 100,000 initializations (which would be slow), and for little benefit (since you’re overwriting those values anyway). For now, **you should always initialize your variables because the cost of doing so is minuscule compared to the benefit.** Once you are more comfortable with the language, there may be certain cases where you omit the initialization for optimization purposes. But this should always be done selectively and intentionally.

### 3.7.3 *WARNING: Debug mode initializes with preset values*

Some compilers, such as Visual Studio, *will* initialize the contents of memory to some preset value when you're using a debug build configuration. This will not happen when using a release build configuration.

### 3.7.4 *Undefined behavior*

Using the value from an uninitialized variable is our first example of undefined behavior. **Undefined behavior** (often abbreviated **UB**) is the result of executing code whose behavior is not well-defined by the C++ language. In this case, the C++ language doesn't have any rules determining what happens if you use the value of a variable that has not been given a known value. Consequently, if you actually do this, undefined behavior will result.

### 3.7.5 *Implementation-defined behavior*

**Implementation-defined behavior** means the behavior of some syntax is **left up to the implementation (the compiler) to define**. Such behaviors must be consistent and documented, but different compilers may produce different results.

Example:

<code>sizeof(int) // the number of bytes assigned to an integer type depends on the compiler</code>
---

### 3.7.6 *Unspecified behavior*

This is almost identical to implementation-defined behavior. The only difference here is that the **implementation (compiler) doesn't have to document why it behaves a certain way**.

### 3.7.7 *Best practice*

Avoid implementation-defined and unspecified behavior whenever possible, as they may cause your program to malfunction on other implementations.

### 3.8 Keywords and naming identifiers

#### 3.8.1 Keywords

A - C	D - P	R - Z
alignas (C++11)	decltype (C++11)	constexpr (reflection TS)
alignof (C++11)	default (1)	register (2)
and	delete (1)	reinterpret_cast
and_eq	do	requires (C++20)
asm	double	return
atomic_cancel (TMTS)	dynamic_cast	short
atomic_commit (TMTS)	else	signed
atomic_noexcept (TMTS)	enum (1)	sizeof (1)
auto (1) (2) (3) (4)	explicit	static
bitand	export (1) (3)	static_assert (C++11)
bitor	extern (1)	static_cast
bool	false	struct (1)
break	float	switch
case	for (1)	synchronized (TMTS)
catch	friend	template
char	goto	this (4)
char8_t (C++20)	if (2) (4)	thread_local (C++11)
char16_t (C++11)	inline (1)	throw
char32_t (C++11)	int	true
class (1)	long	try
compl	mutable (1)	typedef
concept (C++20)	namespace	typeid
const	new	typename
constexpr (C++11)	noexcept (C++11)	union
constinit (C++20)	not	unsigned
const_cast	not_eq	using (1)
continue	nullptr (C++11)	virtual
co_await (C++20)	operator (4)	void
co_return (C++20)	or	volatile
co_yield (C++20)	or_eq	wchar_t
	private (3)	while
	protected	xor
	public	xor_eq

- (1) — meaning changed or new meaning added in C++11.
- (2) — meaning changed or new meaning added in C++17.
- (3) — meaning changed or new meaning added in C++20.
- (4) — new meaning added in C++23.

### 3.8.2 Identifiers

#### 3.8.2.1 Naming rules

Now that you know how you can name a variable, let's talk about how you should name a variable (or function). First, it is a convention in C++ that **variable names should begin with a lowercase letter**. If the variable name is a **single word or acronym, the whole thing should be written in lowercase letters**.

- Names **must begin with a letter or an underscore**
- Names **cannot contain whitespaces or special characters** like !, #, %, etc.
- **Reserved words** (like C++ keywords, such as int) **cannot be used** as names
- Note: names are case-sensitive

#### 3.8.2.2 Beste practice:

When working in an existing program, use the conventions of that program (even if they don't conform to modern best practices). Use modern best practices when you're writing new programs. In any case, **avoid abbreviations (unless they are common/unambiguous)**. Although they reduce the time you need to write your code, they make your code harder to read.

### 3.9 Whitespace and basic formatting

**Whitespace** is a term that refers to characters that are used for formatting purposes. In C++, this refers primarily to **spaces, tabs, and newlines**.

Whitespace in C++ is generally **used for 3 things**:

- Separating certain language elements
- Inside text
- Formatting code

#### 3.9.1.1 Whitespace rules

- **Separate language elements** such as keywords and identifiers
- **Comments** on a newline have to be re-assigned to be comments again: `//`
- **Preprocessor** using directives must be **on separate lines**
- Quoted text
  - **Spaces** in between tags **are taken literally** by the compiler
  - **Newlines** within text is only allowed in a certain format, just pressing enter in text such as a string is **not allowed**.

```
int x; // int and x must be whitespace separated
```

```
int
```

```
x; // this is also valid
```

The whitespace between the **keyword** `int` and the **identifier** `x` makes it so that the compiler doesn't see `intx`, it would see `intx` as an identifier. It doesn't matter how much whitespace there is between them as the compiler will strip this out



```
int main()
{
    std::cout << "Hello
    World!\n";

    return 0;
}
```

← Incorrect newline implementation

Error List		
Entire Solution		
13 Errors 0 Warnings 0		
	Code	Description
✗	C2065	'n': undeclared identifier
✗	C2065	'World': undeclared identifier
abc	E0065	expected a ';'
✗	C2017	illegal escape sequence
abc	E0008	missing closing quote
abc	E0008	missing closing quote
✗	C2001	newline in constant
✗	C2001	newline in constant
✗	C2143	syntax error: missing ';' before '!'
✗	C2143	syntax error: missing ';' before 'return'
✗	C2143	syntax error: missing ';' before 'string'
✗	C2146	syntax error: missing ';' before identifier 'World'
abc	E0007	unrecognized token

← Error output

```
int main()
{
    std::cout << "Hello"
               << "World!\n";

    return 0;
}
```

← Correct newline implementation

### 3.9.1.2 Using whitespace to format code

Whitespace is otherwise generally ignored. This means we can use whitespace wherever we like to format our code in order to make it easier to read.

**Examples: less readable → more readable**

```
1 | #include <iostream>
2 | int main(){std::cout<<"Hello world";return 0;}
```

```
1 | #include <iostream>
2 | int main() {
3 |     std::cout << "Hello world";
4 |     return 0;
5 | }
```

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Hello world";
6 |
7 |     return 0;
8 | }
```

### 3.9.1.3 Basic formatting

Unlike some other languages, **C++ does not enforce any kind of formatting restrictions** on the programmer. For this reason, we say that **C++ is a whitespace-independent language**.

This is a mixed blessing. On one hand, it's nice to have the freedom to do whatever you like. On the other hand, many different methods of formatting C++ programs have been developed throughout the years, and you will find (sometimes significant and distracting) disagreement on which ones are best. **Our basic rule of thumb is that the best styles are the ones that produce the most readable code, and provide the most consistency.**

### 3.9.1.4 Recommendations

1. It's fine to use **either tabs or spaces** for indentation (most IDEs have a setting where you can convert a tab press into the appropriate number of spaces).

Developers who prefer spaces tend to do so because it ensures that code is precisely aligned as intended regardless of which editor or settings are used.

Proponents of using tabs wonder why you wouldn't use the character designed to do indentation for indentation, especially as you can set the width to whatever your personal preference is.

2. There are **two conventional styles** for function braces each with their own variations

**K & R** or **C-style**, puts the opening bracket on the same line as the statement

**Reduces the amount of vertical whitespace** (as you aren't devoting an entire line to an opening curly brace), so you can fit more code on a screen. This enhances code comprehension, as you don't need to scroll as much to understand what the code is doing.

```
int main(){
    std::cout << "Hello World!\n";

    return 0;
}
```

**Allman**, put the opening bracket on a separate line right under the statement

This **enhances readability**, and is **less error prone** since your brace pairs should always be indented at the same level. If you get a compiler error due to a brace mismatch, it's very easy to see where the issue came from.

```
int main()
{
    std::cout << "Hello World!\n";

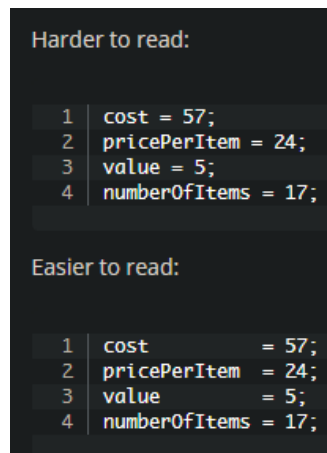
    return 0;
}
```

- Each statement within curly braces should start one tab in from the opening brace of the function it belongs to
- Lines should not be too long. Typically, 80 characters** has been the de facto standard for the maximum length a line should be. If a line is going to be longer, it should be split (**at a reasonable spot**) into multiple lines.
- If a long line is **split** with an operator (e.g. << or +), **the operator should be placed at the beginning of the next line**, not the end of the current line.

```
int main()
{
    std::cout << "Hello"
               << "World!\n";

    return 0;
}
```

6. Use whitespace to make your code easier to read by aligning values or comments or adding spacing between blocks of code.



The image shows a comparison of two code snippets. The top snippet, labeled 'Harder to read:', shows four lines of code where the variable names are left-aligned and the values are not aligned. The bottom snippet, labeled 'Easier to read:', shows the same four lines of code but with the values aligned to the right, making the code more readable.

```
Harder to read:
1 | cost = 57;
2 | pricePerItem = 24;
3 | value = 5;
4 | numberOfItems = 17;

Easier to read:
1 | cost      = 57;
2 | pricePerItem = 24;
3 | value     = 5;
4 | numberOfItems = 17;
```

7. Use the built-in automatic formatting feature for consistency, if available.

### 3.9.1.5 Style guides

A **style guide** is a concise, opinionated document containing (sometimes arbitrary) programming conventions, formatting guidelines, and best practices. The goal of a style guide is to ensure that all developers on a project are programming in a consistent manner.

- C++ Core Guidelines by Bjarne Stroustrup and Herb Sutter
- Google
- LLVM
- GCC / GNU

We generally **favor the C++ Core Guidelines**, as they are up to date and widely applicable.

### 3.10 Introduction to literals

Consider the following two statements:

```
int main()
{
    std::cout << "Hello world!";
    int x{ 5 };

    return 0;
}
```

“Hello world!” and ‘5’ are **literal constants**. They are fixed values inserted directly into the source code. Literals and variables both have a value (and a type). Unlike a variable (whose value can be set and changed through initialization and assignment respectively), **the value of a literal is fixed (5 is always 5)**. This is why literals are called constants.

#### 3.10.1 Compiler-related

```
3  int main()
4  {
5      std::cout << 5 << '\n'; // print the value of a literal
6
7      int x{ 5 };
8      std::cout << x << '\n'; // print the value of a variable
9      return 0;
10 }
```

On line 7, we’re creating a variable named `x`, and initializing it with value 5. **The compiler will generate code that copies the literal value 5 into whatever memory location is given to `x`.**

On line 8, **when we print `x`, the compiler will generate code that causes `std::cout` to print the value at the memory location of `x` (which has value 5).**

Thus, both output statements do the same thing (print the value 5). But **in the case of the literal, the value 5 can be printed directly**. In the case of the variable, the value 5 must be fetched from the memory the variable represents. This also explains why a literal is constant while a variable can be changed. A literal’s value is placed directly in the executable, and the executable itself can’t be changed after it is created. A variable’s value is placed in memory, and the value of memory can be changed while the executable is running.

### 3.11 Introduction to Operators

In mathematics, an **operation** is a process involving zero or more input values (called **operands**) that produces a new value (called an *output value*). The specific operation to be performed is denoted by a symbol called an **operator**.

You are likely already quite familiar with standard arithmetic operators from common usage in mathematics, including addition (+), subtraction (-), multiplication (\*), and division (/). In C++, assignment (=) is an operator as well, as are insertion (<<), extraction (>>), and equality (==). While most operators have symbols for names (e.g. +, or ==), there are **also a number of operators that are keywords** (e.g. **new**, **delete**, and **throw**).

The number of operands that an operator takes as input is called the operator's **arity**. Few people know what this word means, so don't drop it in a conversation and expect anybody to have any idea what you're talking about. **Operators in C++ come in four different arities:**

- **Unary**, acts on **one** operand  
An example of a unary operator is the - operator. For example, given -5, operator- takes literal operand 5 and flips its sign to produce new output value -5.
- **Binary**, acts on **two** operands (often named the left and right operand)  
An example of a binary operator is the + operator. For example, given 3 + 4, operator+ takes the left operand 3 and the right operand 4 and applies mathematical addition to produce new output value 7. The insertion (<<) and extraction (>>) operators are binary operators, taking std::cout or std::cin on the left side, and the value to output or variable to input to on the right side.
- **Ternary**, acts on **three** operands  
There is only one of these in C++ (the conditional operator), which we'll cover later.
- **Nullary**, acts on **zero** operands  
There is also only one of these in C++ (the throw operator), which we'll also cover later.

Note that some operators have more than one meaning depending on how they are used. For example, operator- has two contexts. It can be used in unary form to invert a number's sign (e.g. to convert 5 to -5, or vice versa), or it can be used in binary form to do subtraction (e.g. 4 - 3).



### 3.11.1 Chaining operators

Operators can be **chained together such that the output of one operator can be used as the input for another operator**. For example, given the following: `2 * 3 + 4`, the multiplication operator goes first, and converts left operand 2 and right operand 3 into return value 6 (which becomes the left operand for the plus operator). Next, the plus operator executes, and converts left operand 6 and right operand 4 into new value 10.

We'll talk more about the order in which operators execute when we do a deep dive into the topic of operators. For now, it's enough to know that the arithmetic operators execute in the same order as they do in standard mathematics: **Parenthesis first, then Exponents, then Multiplication & Division, then Addition & Subtraction**. This ordering is sometimes abbreviated ***PEMDAS***, or expanded to the mnemonic "Please Excuse My Dear Aunt Sally".

### 3.11.2 Return values and side effects

Most operators in C++ just use their operands to calculate a return value. There are a few operators that do not produce return values (such as `delete` and `throw`). We'll cover what these do later. Some operators have additional behaviors. An operator (or function) that has some observable effect beyond producing a return value is said to have a **side effect**.

For example, when `x = 5` is evaluated, **the assignment operator has the side effect of assigning the value 5 to variable x**. The changed value of `x` is observable (e.g. by printing the value of `x`) even after the operator has finished executing. `std::cout << 5` **has the side effect of printing 5 to the console**. We can observe the fact that 5 has been printed to the console even after `std::cout << 5` has finished executing.

**Operators with side effects are usually called for the behavior of the side effect rather than for the return value** (if any) those operators produce.

### 3.11.3 Nomenclature

In common language, the term “side effect” is typically used to mean a secondary (often negative or unexpected) result of some other thing happening (such as taking medicine). For example, a common side effect of taking oral antibiotics is diarrhea. As such, we often think of side effects as things we want to avoid, or things that are incidental to the primary goal.

In C++, **the term “side effect” has a different meaning: it is an observable effect of an operator or function beyond producing a return value.**

Since assignment has the observable effect of changing the value of an object, this is considered a side effect. We use certain operators (e.g. the assignment operator) primarily for their side effects. In such cases, the side effect is both beneficial and predictable (and it is the return value that is often incidental).

### 3.12 Introduction to expressions

An **expression** is a non-empty **sequence of literals, variables, operators, and function calls** that calculates a single value. The process of executing an expression is called **evaluation**, and the single value produced is called the **result** of the expression. When an expression is evaluated, **each of the terms inside the expression are evaluated**, until a single value remains.

While most expressions are used to calculate a value, **expressions can also identify an object** (which can be evaluated to get the value held by the object) or a function.

Example:

```
2           // 2 is a literal that evaluates to value 2
"Hello world!" // "Hello world!" is a literal that evaluates to text "Hello world!"
x           // x is a variable that evaluates to the value of x
2 + 3       // operator+ uses operands 2 and 3 to evaluate to value 5
five()      // evaluates to the return value of function five()
```

As you can see, literals evaluate to their own values. Variables evaluate to the value of the variable. Operators (such as **operator+**) use their operands to evaluate to some other value. We haven't covered function calls yet, but in the context of an expression, function calls evaluate to whatever value the function returns. **Expressions involving operators with side effects are a little trickier:**

```
x = 5       // x = 5 has side effect of assigning 5 to x, evaluates to x
x = 2 + 3    // has side effect of assigning 5 to x, evaluates to x
std::cout << x // has side effect of printing value of x to console, evaluates to std::cout
```

Note that **expressions do not end in a semicolon**, and **cannot be compiled by themselves**. For example, if you were to try compiling the expression `x = 5`, your compiler would complain (probably about a missing semicolon). Rather, expressions are always evaluated as part of statements.

If you were to break `int x {2+3};` down into syntax, it would look like this:

```
type identifier {expression};
```

**Type** could be any valid type (we chose `int`). **identifier** could be any valid name (we chose `x`). And **expression** could be any valid expression (we chose `2 + 3`, which uses two literals and an operator).

**Key insight:** Wherever you can use a single value in C++, you can use a value-producing expression instead, and the expression will be evaluated to produce a single value.

### 3.12.1 *Useless expression statements*

We can also make expression statements that compile but have no effect. For example, the expression statement **(2 \* 3;)** is an expression statement whose **expression evaluates to the result value of 6, which is then discarded**. While **syntactically valid**, such expression statements are useless. **Some compilers (such as gcc and Clang) will produce warnings** if they can detect that an expression statement is useless.

### 3.12.2 *Subexpressions, full expressions and compound expressions*

Simplifying a bit, a **subexpression** is an expression used as an operand. For example, the **subexpressions of  $x = 4 + 5$**  are  **$x$**  and  **$4 + 5$** . The **subexpressions of  $4 + 5$**  are **4** and **5**.

A **full expression** is an expression that is not a subexpression.  **$x = 4 + 5$**  is the full expression.

In casual language, a **compound expression** is an expression that contains two or more uses of operators.  **$x = 4 + 5$**  is a **compound expression** because it contains **two uses of operators** (**operator=** and **operator+**)

### 3.12.3 *Recap*

What is the difference between a statement and an expression?

**Statements** are used when we want the program to **perform an action**.

**Expressions** are used when we want the program to **calculate a value**.

### 3.13 Chapter summary

A **statement** is a type of instruction that causes the program to perform some action. Statements are often terminated by a semicolon.

A **function** is a collection of statements that execute sequentially. Every C++ program must include a special function named `main`. When you run your program, execution starts at the top of the `main` function.

In programming, the name of a function (or object, type, template, etc.) is called its **identifier**.

An **object** is a **region of storage** (usually memory) that can store a value, and has other associated properties. In C++, **direct memory access is discouraged**. Instead, **we access memory indirectly through an object**.

A **variable is an object** that has been named (identifier). Naming our objects let us refer to them again later in the program.

To create a variable, we use a **definition statement**. Later we can **assign** it a value. When the program is run, each defined variable is **instantiated**, which means it is assigned a memory address. The process of **specifying an initial value** for an object is called **initialization**, and the syntax used to initialize an object is called an **initializer**.

The rules that govern how elements of the C++ language are constructed is called **syntax**. A **syntax error** occurs when you violate the grammatical rules of the language.

Every C++ program must have a special function named `main` (all lower - case letters) which is of the `int` type. (because the number being returned is the exit status code of `main`)

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

**Comments** allow the programmer to leave notes in the code. C++ supports two types of comments. Line comments start with a `//` and run to the end of the line. Block comments start with a `/*` and go to the paired `*/` symbol. Don't nest block comments.

**Data** is any information that can be moved, processed, or stored by a computer. A single piece of data is called a **value**.

A **data type** tells the compiler how to interpret a piece of data into a meaningful value. An **integer** is a number that can be written without a fractional component.

Initialization Type	Example	Note
Default initialization	<code>int x;</code>	In most cases, leaves variable with indeterminate value
Copy initialization	<code>int x = 5;</code>	
Direct initialization	<code>int x ( 5 );</code>	
Direct list initialization	<code>int x { 5 };</code>	Narrowing conversions disallowed
Copy list initialization	<code>int x = { 5 };</code>	Narrowing conversions disallowed
Value initialization	<code>int x {};</code>	Usually performs zero initialization

**Direct initialization** is sometimes called **parenthesis initialization**.

**List initialization (including value initialization)** is sometimes called **uniform initialization** or **brace initialization**. You should prefer brace initialization over the other initialization forms, and prefer initialization over assignment.

A **literal constant** is a fixed value inserted directly into the source code. Examples are 5 and “Hello world!”.

An **operation** is a process involving zero or more input values, called **operands**. The specific operation to be performed is denoted by the provided **operator**. The result of an operation produces an output value.

**Unary** operators take one operand. **Binary** operators take two operands, often called left and right. **Ternary** operators take three operands. **Nullary** operators take zero operands.

An **expression** is a sequence of literals, variables, operators, and function calls that are evaluated to produce a single output value. The calculation of this output value is called **evaluation**. The value produced is the **result** of the expression.

An **expression statement** is an expression that has been turned into a statement by placing a semicolon at the end of the expression.

**When writing programs, add a few lines or a function, compile, resolve any errors, and make sure it works. Don’t wait until you’ve written an entire program before compiling it for the first time!**

First-draft programs are often messy and imperfect. Most code requires cleanup and refinement to get great!

## 4 Functions and files

### 4.1 Introduction to functions

In the last chapter, we defined a function as a **collection of statements** that execute sequentially. While that is certainly true, that definition doesn't provide much insight into why functions are useful. Let's update our definition: A function is a **reusable sequence of statements** designed to do a particular job.

You already know that every executable program must have a function named `main` (which is where the program starts execution when it is run). However, **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**.

A program will be executing statements sequentially inside one function when it encounters a **function call**. A function call is an expression that tells the CPU to **interrupt the current function and execute the called function**. The CPU **"puts a bookmark"** at the current point of execution, and then **calls** (executes) the function named in the function call. When the called function ends, **the CPU returns back to the point it bookmarked**, and resumes execution.

The function initiating the function call is the **caller**, and the function being called is the **callee** or **called** function.

#### 4.1.1 *User-defined functions*

Functions that you write yourself are called **user-defined functions**.

The start of a user-defined function is informally called the **function header**, and it tells the compiler about the existence of a function, the function's name (**identifier**), and some other information that we'll cover in future lessons (like the return type and parameter types).

The **parentheses** after the identifier tell the compiler that we're **defining a function**.

The curly braces and statements in-between are called the **function body**. This is where the statements that determine what your function does will go.

```
4 void PrintTheSumOf(int a, int b)
5 {
6     // Great, we just landed here from main()
7     // Now let's execute this sequentially
8
9     std::cout << "Inside of PrintTheSumOf()" << "\n";
10    // Ok here we print some text to show we are inside of this function
11
12
13    int sum{ a + b };
14    // Here we initialize a variable (named object) with the integer type
15    // It's a good thing that we assign it a value as to avoid undefined behaviour
16    // In this case the name is sum and it holds the expression a operator+ b (our initializer)
17    // Now or compiler will instantiate it!
18
19    std::cout << sum << "\n";
20    // This statement is little more complicated
21    // We use cout and operator<< to display the sum and then add a new line
22
23    std::cout << "Ending of PrintTheSumOf()" << std::endl;
24 }
25
26
27 int main()
28 {
29     // Because this is main, this is the actual start of the program
30
31     PrintTheSumOf(3, 7);
32     // We encounter a function, we know this because of the parenthesis
33     // It seems to be doing something with two literal numbers
34     // Good thing it has a name (identifier), let's jump there
35     // But first we take note of this location so it's easy to jump back when we're done
36     // READ PrintTheSumOf()
37
38     std::cout << "back inside of main()" << std::endl;
39
40     return 0;
41     // This is the absolute end of our program
42     // Because main is a special function there is a lot going on
43     // The literal 0 here mean the program can exit with success
44     // The literal 1 would mean the program exits with failure
45 }
46
```



#### 4.1.1.1 Functions can call functions that call other functions

You've already seen that function **main** can call another function. Any function can call any other function. In the following program, function **main** calls function **doA**, which calls function **doB**:

```
1  #include <iostream> // for std::cout
2
3  void doB()
4  {
5      std::cout << "In doB()\n";
6  }
7
8
9  void doA()
10 {
11     std::cout << "Starting doA()\n";
12
13     doB();
14
15     std::cout << "Ending doA()\n";
16 }
17
18 // Definition of function main()
19 int main()
20 {
21     std::cout << "Starting main()\n";
22
23     doA();
24
25     std::cout << "Ending main()\n";
26
27     return 0;
28 }
```

#### 4.1.1.2 Nested functions are not supported

Unlike some other programming languages, in C++, functions cannot be defined inside other functions. The following program is not legal:

```
1  #include <iostream>
2
3  int main()
4  {
5      void foo() // Illegal: this function is nested inside function main()
6      {
7          std::cout << "foo!\n";
8      }
9
10     foo(); // function call to foo()
11     return 0;
12 }
```

The proper way to write the above program is:

```
1  #include <iostream>
2
3  void foo() // no longer inside of main()
4  {
5      std::cout << "foo!\n";
6  }
7
8  int main()
9  {
10     foo();
11     return 0;
12 }
```

#### 4.1.1.3 As an aside...

“foo” is a meaningless word that is often used as a **placeholder name for a function or variable** when the name is unimportant to the **demonstration of some concept**. Such words are called **metasyntactic variables** (though in common language they’re often called “**placeholder names**” since nobody can remember the term “metasyntactic variable”). Others include: bar, baz and 3-letter words that end in “oo”, such as goo, moo and boo.

For those interested in etymology (how words evolve), RFC 3092 is an interesting read.

## 4.2 Function return values (value-returning functions)

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. Note that this doesn't determine what specific value is returned -- it only determines what type of value will be returned. Void is a special case that doesn't have to return anything. We explore functions that return void further in the next lesson.

**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 specific value returned from a function is called the **return value**. When the return statement is executed, the function exits immediately, and the return value is **copied** from the function back to the caller. **This process is called return by value.**

When a called function returns a value, **the caller may decide to use that value in an expression or statement** (e.g. by using it to initialize a variable, or sending it to `std::cout`) or ignore it (by doing nothing else). If the caller ignores the return value, it is discarded (nothing is done with it).

```

1  #include <iostream>
2
3  int getValueFromUser() // this function now returns an integer value
4  {
5      std::cout << "Enter an integer: ";
6      int input{};
7      std::cin >> input;
8
9      return input; // return the value the user entered back to the caller
10 }
11
12 int main()
13 {
14     int num { getValueFromUser() }; // initialize num with the return value of getValueFromUser()
15
16     std::cout << num << " doubled is: " << num * 2 << '\n';
17
18     return 0;
19 }
```

When this program executes, the first statement in `main` will create an `int` variable named `num`. When the program goes to initialize `num`, it will see that there is a function call to `getValueFromUser()`, so it will go execute that function. Function `getValueFromUser`, asks the user to enter a value, and then it returns that value back to the caller (`main`). This return value is used as the initialization value for variable `num`.

### 4.2.1 Revisiting main()

You now have the conceptual tools to understand how the `main()` function actually works. When the program is executed, the operating system makes a function call to `main()`. Execution then jumps to the top of `main()`. The statements in `main()` are executed sequentially. **Finally, `main()` returns an integer value (usually 0), and your program terminates.**

**C++** disallows calling the `main()` function explicitly. **C** does allow `main()` to be called explicitly, so some C++ compilers will allow this for compatibility reasons. For now, you should also **define your `main()` function at the bottom of your code file**, below other functions, and avoid calling it explicitly.

#### 4.2.1.1 Status codes

The **return value** from `main()` is sometimes called a **status code** (also sometimes called an **exit code**, or rarely a return code). Just like the return value of a function is passed back to the caller of the function, the status code is passed back to the caller of the program. The caller of the program can then use this status code to determine whether your program ran successfully or not.

**0:** Indicates that the program ran successfully without any errors. This is the standard value for a successful program termination. Alternatively, we can write **`EXIT_SUCCESS`** (defined in the standard library).

**1:** Indicate that the program encountered some form of error or abnormal termination. Alternatively, we can write **`EXIT_FAILURE`** (defined in the standard library).

```
#include <stdlib.h>

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

#### **4.2.2** *A value-returning function that does not return a value will produce undefined behavior*

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. **A value-returning function must return a value of that type (using a return statement)**, otherwise undefined behavior will result.

In most cases, compilers will detect if you've forgotten to return a value. However, in some complicated cases, the compiler may not be able to properly determine whether your function returns a value or not in all cases, so you should not rely on this.

The only exception to the rule that a value-returning function must return a value via a return statement is for function **main()**. The function **main()** will **implicitly return the value 0 if no return statement is provided**. That said, it is best practice to explicitly return a value from **main**, both to show your intent, and for consistency with other functions (which will exhibit undefined behavior if a return value is not specified).

#### **4.2.3** *Functions can only return a single value*

A value-returning function can only return a single value back to the caller each time it is called. Note that the value provided in a return statement doesn't need to be literal -- it can be the result of any valid expression, including a variable or even a call to another function that returns a value. There are various ways to work around the limitation of functions only being able to return a single value, which we'll cover in future lessons.

#### **4.2.4** *The function author can decide what the return value means*

The meaning of the value returned by a function is determined by the function's author. Some functions use return values as status codes, to indicate whether they succeeded or failed. Other functions return a calculated or selected value. Other functions return nothing.

Because of the wide variety of possibilities here, it's a good idea to document your function with a comment indicating what the return values mean.

#### 4.2.5 Reusing functions

While this program works, it's a little redundant. In fact, this program violates one of the central tenets of good programming: **Don't Repeat Yourself** (often abbreviated **DRY**).

Why is repeated code bad? If we wanted to change the text "Enter an integer:" to something else, we'd have to update it in two locations. And what if we wanted to initialize 10 variables instead of 2? That would be a lot of **redundant code** (making our programs longer and harder to understand), and a lot of **room for typos to creep in**.

```
1 | #include <iostream>
2 |
3 | int getValueFromUser()
4 | {
5 |     std::cout << "Enter an integer: ";
6 |     int input{};
7 |     std::cin >> input;
8 |
9 |     return input;
10 | }
11 |
12 | int main()
13 | {
14 |     int x{ getValueFromUser() }; // first call to getValueFromUser
15 |     int y{ getValueFromUser() }; // second call to getValueFromUser
16 |
17 |     std::cout << x << " + " << y << " = " << x + y << '\n';
18 |
19 |     return 0;
20 | }
```

In this program, we call **getValueFromUser** twice, once to initialize variable **x**, and once to initialize variable **y**. That saves us from duplicating the code to get user input, and reduces the odds of making a mistake. Once we know **getValueFromUser** works, we can call it as many times as we desire.

This is the **essence of modular programming**: the ability to write a function, test it, ensure that it works, and then know that we can reuse it as many times as we want and it will continue to work (so long as we don't modify the function -- at which point we'll have to retest it).

#### 4.2.6 *Best practice*

Follow **DRY**: “**don’t repeat yourself**”. If you need to do something more than once, consider how to modify your code to **remove as much redundancy as possible**. Variables can be used to store the results of calculations that need to be used more than once (so we don’t have to repeat the calculation). Functions can be used to define a sequence of statements we want to execute more than once. And loops (which we’ll cover in a later chapter) can be used to execute a statement more than once.

Like all best practices, **DRY is meant to be a guideline, not an absolute**. Reader Yariv has noted that DRY can harm overall comprehension when code is broken into pieces that are too small.

The opposite of DRY is WET (“Write everything twice”).

### 4.3 Void functions (non-value returning 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. A function that does not return a value is called a **non-value returning function** (or a **void function**). Trying to return a value from a non-value returning function will result in a **compilation error**.

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

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

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

    return 0;
}
```

#### 4.3.1 Void return statement

**A void function will automatically return** to the caller at the end of the function. No return statement is required.

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. This is the same thing that happens at the end of the function anyway. Consequently, putting an empty return statement at the end of a void function is **redundant**.

Best practice:

Do **not** put a return statement at the end of a non-value returning function.



#### 4.4 Introduction to function parameters and arguments

In the previous lesson, we learned that we could have a function return a value back to the function's caller. Now we need a way to **pass values of objects in order to manipulate them in our functions**, this is where parameters and arguments come in.

In many cases, it is useful to be able to pass information to a function being called, so that the function has data to work with. For example, if we wanted to write a function to add two numbers, we need some way to tell the function which two numbers to add when we call it. Otherwise, how would the function know what to add? We do that via function **parameters and 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.

An **argument** is a value that is passed from the caller to the function when a function call is made.

```
✓ #include <iostream>
  [ #include <string>

  ✓ void foo(std::string sentence)
    {
      std::cout << sentence << '\n';
    }

  ✓ int main()
    {
      foo("Hello World!");

      return 0;
    }
  [ ]
```

In the example above we ask for a sentence to be passed of the type `std::string`, the variable `sentence` is the **parameter**. The literal `"Hello World!"` is the **argument** we pass.

#### 4.4.1 How parameters and arguments work together

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**.

Note that the **number of arguments must generally match the number of function parameters**, or the compiler will throw an error. The argument passed to a function can be any valid expression (as the argument is essentially just an initializer for the parameter, and initializers can be any valid expression).

#### 4.4.2 Using return values as arguments

Now we step it up a bit, in previous lesson we discussed return values. We could also use these by passing them to another function.

```
1  #include <iostream>
2
3  int getValueFromUser()
4  {
5      std::cout << "Enter an integer: ";
6      int input{};
7      std::cin >> input;
8
9      return input;
10 }
11
12 void printDouble(int value)
13 {
14     std::cout << value << " doubled is: " << value * 2 << '\n';
15 }
16
17 int main()
18 {
19     printDouble(getValueFromUser());
20
21     return 0;
22 }
```

#### 4.4.3 How parameters and return values work together

By using both parameters and a return value, we can create functions that take data as input, do some calculation with it, and return the value to the caller.

```
1  #include <iostream>
2
3  int add(int x, int y)
4  {
5      return x + y;
6  }
7
8  int main()
9  {
10     std::cout << add(4, 5) << std::endl;
11     return 0;
12 }
```

The diagram illustrates the execution of the `add` function within the `main` function. Annotations show the arguments `x = 4` and `y = 5` being passed to the `add` function. The return value of the function is `9`, which is then printed by the `main` function.

#### 4.4.4 Unreferenced parameters

In certain cases, you will encounter functions that have parameters that are not used in the body of the function. These are called unreferenced parameters.

This can happen when a function parameter was once used, but is not used any longer. Just like with unused local variables, your compiler will probably warn you.

**If the unused function parameter were simply removed, then any existing call to the function would break** (because the function call would be supplying more arguments than the function could accept).

```
1 void doSomething(int count) // warning: unreferenced parameter count
2 {
3     // This function used to do something with count but it is not used any longer
4 }
5
6 int main()
7 {
8     doSomething(4);
9
10    return 0;
11 }
```

In a function definition, the **name of a function parameter is optional**. Therefore, in cases where a function parameter needs to exist but is not used in the body of the function, you can simply omit the name. A parameter without a name is called an **unnamed parameter**.

```
1 void doSomething(int) // ok: unnamed parameter will not generate warning
2 {
3 }
```

Best practice:

When a function parameter **exists but is not used in the body of the function, do not give it a name**. You can optionally put a name inside a comment.

## 4.5 Introduction to local scope

Variables defined inside the body of a function are called **local variables** (as opposed to global variables, which we'll discuss in the future).

### 4.5.1 Lifetime

In Introduction to objects and variables, we discussed how a variable definition such as `int x;` causes the variable to be instantiated (created) when this statement is executed. **Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition.**

```
int add(int x, int y) // x and y created and initialized here
{
    int z{ x + y };    // z created and initialized here

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

The natural follow-up question is, “**so when is an instantiated variable destroyed?**”. 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).

Much like a person's lifetime is defined to be the time between their birth and death, an object's **lifetime** is defined to be the time between its creation and destruction. Note that **variable creation and destruction happen when the program is running** (called **runtime**), not at compile time. Therefore, **lifetime is a runtime property**.

#### 4.5.1.1 What happens when an object is destroyed?

In most cases, **nothing**. The **destroyed object becomes invalid**, and any **further use of the object will result in undefined behavior**. At some point after destruction, the memory used by the object will be freed up for reuse.

Advanced:

If the object is a **class type object**, prior to destruction, a special function called a **destructor** is **invoked**. In many cases, the destructor does nothing, in which case no cost is incurred.

### 4.5.2 Local scope

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 **cannot be seen**, we cannot 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.

A local variable's scope begins at the point of variable definition, and stops at the end of the set of curly braces in which it is defined (or for function parameters, at the end of the function). **This ensures variables cannot be used before the point of definition** (even if the compiler opts to create them before then). Local variables defined in one function are also not in scope in other functions that are called.

#### 4.5.2.1 "Out of scope" vs "going out of scope"

The terms "out of scope" and "going out of scope" can be confusing to new programmers.

An identifier is **out of scope anywhere it cannot be accessed** within the code.

The term "going out of scope" is **typically applied to objects** rather than identifiers. We say an object **goes out of scope at the end of the scope** (the end curly brace) in which the object was instantiated. A local variable's **lifetime ends at the point where it goes out of scope**, so local variables are destroyed at this point. Note that **not all types of variables are destroyed** when they go out of scope.

```

1 | #include <iostream>
2 |
3 | int add(int x, int y) // x and y are created and enter scope here
4 | {
5 |     // x and y are usable only within add()
6 |     return x + y;
7 | } // y and x go out of scope and are destroyed here
8 |
9 | int main()
10 | {
11 |     int a{ 5 }; // a is created, initialized, and enters scope here
12 |     int b{ 6 }; // b is created, initialized, and enters scope here
13 |
14 |     // a and b are usable only within main()
15 |
16 |     std::cout << add(a, b) << '\n'; // calls add(5, 6), where x=5 and y=6
17 |
18 |     return 0;
19 | } // b and a go out of scope and are destroyed here

```

#### 4.5.2.2 Functional separation

Names used for function parameters or variables declared in a function body are only visible within the function that declares them. This means **local variables within a function can be named without regard for the names of variables in other functions**. This helps keep functions independent.

We could have variable `x` in function `add()` and variable `x` in `main()` and they wouldn't interfere because of functional separation.

#### 4.5.2.3 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.

#### 4.5.2.4 As an aside

Due to the limitations of older, more primitive compilers, the C language used to require all local variables be defined at the top of the function. The equivalent C++ program using that style would look like this:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int x{}, y{}, sum{}; // how are these used?
6 |
7 |     std::cout << "Enter an integer: ";
8 |     std::cin >> x;
9 |
10 |    std::cout << "Enter another integer: ";
11 |    std::cin >> y;
12 |
13 |    sum = x + y;
14 |    std::cout << "The sum is: " << sum << '\n';
15 |
16 |    return 0;
17 | }
```

This style is suboptimal for several reasons:

- The intended use of these variables isn't apparent at the point of definition. You have to scan through the entire function to determine where and how each variable is used.
- The intended initialization value may not be available at the top of the function (e.g. we can't initialize sum to its intended value because we don't know the value of x and y yet).
- There may be many lines between a variable's initializer and its first use. If we don't remember what value it was initialized with, we will have to scroll back to the top of the function, which is distracting.

**This restriction was lifted in the C99 language standard.**



### 4.5.3 Introduction to temporary objects

A **temporary object** (also sometimes called an **anonymous object**) is an unnamed object that is created by the compiler to store a value temporarily.

**Temporary objects have no scope at all** (this makes sense, since scope is a property of an identifier, and temporary objects have no identifier).

Temporary objects are **destroyed at the end of the full expression** in which they are created. In the case where a temporary object is used to initialize a variable, the **initialization happens before the destruction of the temporary**.

Example:

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 the next function that needs it.

### 4.6 Why functions are useful, and how to use them effectively

This chapter I'm not going into detail as it's been hinted at before why functions are useful. However here is a quick summary:

- Organization
- Reusability
- Testing
- Extensibility
- Abstraction

**New programmers often combine calculating a value and printing the calculated value into a single function.** However, **this violates the "one task" rule of thumb for functions**

A function that calculates a value should return the value to the caller and let the caller decide what to do with the calculated value (such as call another function to print the value).

Some others might name this as **separation of concerns**.

## 4.7 Forward declarations and definitions

Take a look at this seemingly innocent sample program:

```
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 | }
```

**Output:** add.cpp(5) : error C3861: 'add': identifier not found

The reason this program doesn't compile is because the **compiler compiles** the contents of code files **sequentially**. This means **top to bottom**.

When the compiler reaches the function call to `add` on line 5 of `main`, it doesn't know what `add` is, because we haven't defined `add` until line 9! That produces the error, identifier not found.

This is somewhat misleading, given that `add` wasn't ever defined in the first place. Despite this, it's useful to generally note that it is fairly common for a single error to produce many redundant or related errors or warnings. It can sometimes be hard to tell whether any error or warning beyond the first is a consequence of the first issue, or whether it is an independent issue that needs to be resolved separately.

**Best practice:** When addressing compilation errors or warnings in your programs, **resolve the first issue** listed and then compile again.

To fix this problem, we need to address the fact that the compiler doesn't know what `add` is.

**There are two common ways to address the issue.**

- Reorder the function definitions
- **Use a forward declaration**

We might not always have the option available to reorder in a sequential manner.

#### 4.7.1 Forward declaration

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.

To write a forward declaration for a function, we use a **function declaration statement** (also called a **function prototype**). The function declaration consists of the **function's return type, name, and parameter types, terminated with a semicolon**. The names of the parameters can be optionally included. The function body is not included in the declaration.

```
1 #include <iostream>
2
3 int add(int x, int y); // forward declaration of add() (using a function declaration)
4
5 int main()
6 {
7     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // this works because we forward declared add() above
8     return 0;
9 }
10
11 int add(int x, int y) // even though the body of add() isn't defined until here
12 {
13     return x + y;
14 }
```

It is worth noting that function declarations **do not need to specify the names of the parameters** (as they are not considered to be part of the function declaration).

**Beste practice:** Keep the parameter names in your function declarations.

#### 4.7.2 *Why forward declarations?*

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.

#### 4.7.3 *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**.

#### 4.7.4 Identifier declaration

Forward declarations are most often used with functions. However, **forward declarations can also be used with other identifiers** in C++, **such as variables and types**. Variables and types have a different syntax for forward declaration, so we'll cover these in future lessons.

#### 4.7.5 Declarations vs. definitions

In C++, you'll frequently hear the words "**declaration**" and "**definition**" used, and often interchangeably. What do they mean? You now have enough fundamental knowledge to understand the difference between the two.

A **declaration** tells the compiler about the existence of an identifier and its associated type information.

A **definition** is a declaration that actually implements (for functions and types) or instantiates (for variables) the identifier.

In C++, **all definitions are declarations**. Conversely, **not all declarations are definitions**. Declarations that aren't definitions are called **pure declarations**. Types of pure declarations include forward declarations for function, variables, and types.

Term	Technical Meaning	Examples
Declaration	Tells compiler about an identifier and its associated type information.	<code>void foo();</code> // function forward declaration (no body) <code>void goo() {};</code> // function definition (has body) <code>int x;</code> // variable definition
Definition	Implements a function or instantiates a variable. Definitions are also declarations.	<code>void foo() { };</code> // function definition (has body) <code>int x;</code> // variable definition
Pure declaration	A declaration that isn't a definition.	<code>void foo();</code> // function forward declaration (no body)
Initialization	Provides an initial value for a defined object.	<code>int x { 2 };</code> // 2 is the initializer

#### 4.7.6 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:

- 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.
- 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: Internal linkage).
- **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.

```
1 | int add(int x, int y)
2 | {
3 |     return x + y;
4 | }
5 |
6 | int add(int x, int y) // violation of ODR, we've already defined function add(int, int)
7 | {
8 |     return x + y;
9 | }
10 |
11 | int main()
12 | {
13 |     int x{};
14 |     int x{ 5 }; // violation of ODR, we've already defined x
15 | }
```

## 4.8 Programs with multiple code files

As programs get larger, it is common to split them into multiple files for organizational or reusability purposes. One advantage of working with an IDE is that they make working with multiple files much easier. You already know how to create and compile single-file projects. Adding new files to existing projects is very easy.

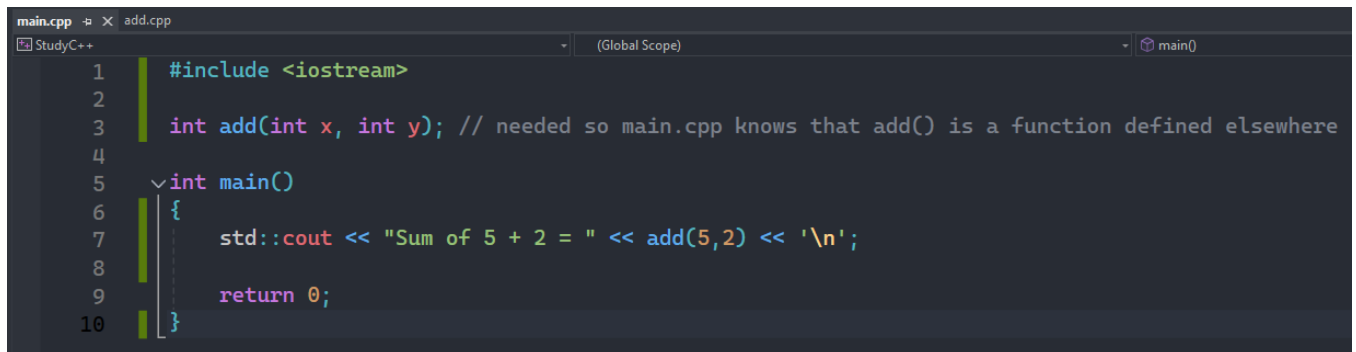
Let's say we **define** the function `add(int x, int y)` in a **separate file** named `add.cpp`. For context we'll use this in the **main function inside of our main.cpp file**.

Your compiler **may compile either** `add.cpp` or `main.cpp` **first**. Either way, `main.cpp` will **fail to compile**. The reason is **it doesn't know what identifier add is**.

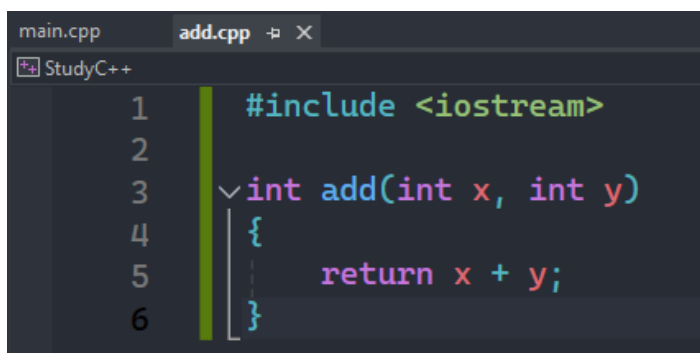
Remember, the compiler **compiles each file individually**. It does not know about the contents of other code files, or remember anything it has seen from previously compiled code files. So even though the compiler may have seen the definition of function `add` previously (if it compiled `add.cpp` first), **it doesn't remember**.

This **limited visibility and short memory is intentional**, for a few reasons:

- It allows the source files of a project to be **compiled in any order**.
- When we change a source file, **only that source file needs to be recompiled**.
- It **reduces the possibility of naming conflicts** between identifiers in different files.



```
main.cpp x add.cpp
StudyC++ (Global Scope) main()
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 << "Sum of 5 + 2 = " << add(5,2) << '\n';
8
9     return 0;
10 }
```



```
main.cpp add.cpp x
StudyC++
1 #include <iostream>
2
3 int add(int x, int y)
4 {
5     return x + y;
6 }
```

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.**

Using this method, we can give files access to functions that live in another file.



## 4.9 Naming collisions and an introduction to namespaces

Let's say you are driving to a friend's house for the first time, and the address given to you is 245 Front Street in Mill City. Upon reaching Mill City, you take out your map, only to discover that **Mill City actually has two different Front Streets across town from each other!** Which one would you go to? Unless there were some additional clues to help you decide (e.g. you remember your friend's house is near the river) you'd have to call your friend and ask for more information. Because this would be **confusing and inefficient** (particularly for your mail carrier), in most countries, all street names and house addresses within a city are **required to be unique**.

Similarly, **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**.

Most naming collisions occur in two cases:

- Two (or more) identically named functions (or global variables) are introduced into **separate files** belonging to the same program. This will result in a **linker error**, as shown above.
- Two (or more) identically named functions (or global variables) are introduced into the **same file**. This will result in a **compiler error**.

As programs get larger and use more identifiers, the odds of a naming collision being introduced increases significantly. The good news is that **C++ provides plenty of mechanisms for avoiding naming collisions**. **Local scope**, which keeps local variables defined inside functions from conflicting with each other, is one such mechanism. **But local scope doesn't work for function names. So how do we keep function names from conflicting with each other?**

#### 4.9.1 *Scope regions*

Back to our address analogy for a moment, having two Front Streets was only problematic because those streets existed within the same city. On the other hand, if you had to deliver mail to two addresses, one at 245 Front Street in Mill City, and another address at 245 Front Street in Jonesville, there would be no confusion about where to go. Put another way, cities **provide groupings that allow us to disambiguate** addresses that might otherwise conflict with each other.

A **scope region** is an **area of source code** where **all declared identifiers are considered distinct from names declared in other scopes** (much like the cities in our analogy). Two identifiers with the same name can be declared in separate scope regions without causing a naming conflict. However, within a given scope region, all identifiers must be unique, otherwise a naming collision will result.

##### 4.9.1.1 **Function body**

The **body of a function is one example of a scope region**. Two identically-named identifiers can be defined in separate functions without issue -- because each function provides a separate scope region, there is no collision. However, if you try to define two identically-named identifiers within the same function, a naming collision will result, and the compiler will complain.

##### 4.9.1.2 **Namespaces**

A **namespace** provides **another type of scope region** (called **namespace scope**) that allows you to declare names inside of it for the purpose of disambiguation. Any names declared inside the namespace won't be mistaken for identical names in other scopes.

**A name declared in a scope region (such as a namespace) won't be mistaken for an identical name declared in another scope.**

**Unlike functions** (which are designed to contain **executable statements**), **only declarations and definitions can appear in the scope of a namespace**. For example, two identically named functions can be defined inside separate namespaces, and no naming collision will occur.

**Key insight:** Only declarations and definitions can appear in the scope of a namespace (not executable statements). However, a function can be defined inside a namespace, and that function can contain executable statements.

#### 4.9.1.3 The global namespace

In C++, **any name** that is **not defined inside a class, function, or a namespace** is considered to be part of an implicitly-defined namespace called the **global namespace** (sometimes also called the **global scope**).

We discuss the global namespace in more detail later on.

For now, there are **two things** you should know:

- Identifiers declared inside the global scope are **in scope from the point of declaration to the end of the file**.
- Although variables **can be defined in the global namespace, this should generally be avoided**.

#### 4.9.1.4 The std namespace

When C++ was originally designed, all of the identifiers in the C++ standard library (including `std::cin` and `std::cout`) were available to be used without the `std::` prefix (they were part of the global namespace).

However, this meant that any identifier in the standard library could potentially conflict with any name you picked for your own identifiers (also defined in the global namespace). Code that was working might suddenly have a naming conflict when you `#included` a new file from the standard library. Or worse, programs that would compile under one version of C++ might not compile under a future version of C++, as new identifiers introduced into the standard library could have a naming conflict with already written code.

So, **C++ moved all of the functionality in the standard library** into a namespace **named std** (short for “**standard**”).

It turns out that `std::cout`'s name isn't really `std::cout`. It's actually just `cout`, and `std` is the **name of the namespace that identifier cout is part of**. Because `cout` is defined in the `std` namespace, the name `cout` won't conflict with any objects or functions named `cout` that we create outside of the `std` namespace (such as in the global namespace).

When accessing an identifier that is defined in a namespace (e.g. `std::cout`), you need to tell the compiler that we're looking for an identifier defined inside the namespace (`std`).

#### 4.9.1.4.1 *Explicit namespace qualifier std::*

The most straightforward way to tell the compiler that we want to use `cout` from the `std` namespace is by explicitly using the **`std::` prefix**.

The `::` symbol is an operator called the **scope resolution operator**. The identifier to the **left** of the `::` symbol **identifies the namespace** that the name to the right of the `::` symbol is contained within. If no identifier to the left of the `::` symbol is provided, the global namespace is assumed.

When an identifier includes a namespace prefix, the identifier is called a **qualified name**.

**Best practice:** Do this. Explicitly use the scope resolution operator to tell the compiler where the identifier is and avoid naming collisions.

#### 4.9.1.4.2 *Using namespace std*

Another way to access identifiers inside a namespace is to use a **using-directive statement**.

A **using directive** allows us to **access the names in a namespace without using a namespace prefix**.

Many texts, tutorials, and even some IDEs recommend or use a using-directive at the top of the program. However, used in this way, **this is a bad practice, and highly discouraged**.

When using a using-directive in this manner, any identifier we define may conflict with any identically named identifier in the `std` namespace. Even worse, while an identifier name may not conflict today, it may conflict with new identifiers added to the `std` namespace in future language revisions. **This was the whole point of moving all of the identifiers in the standard library into the `std` namespace in the first place!**

#### 4.9.2 *Curly braces and indented code*

In C++, curly braces are **often used to delineate a scope region** that is nested within another scope region (braces are also used for some non-scope-related purposes, such as list initialization).

For example, a function defined inside the global scope region uses curly braces to separate the scope region of the function from the global scope.

In certain cases, identifiers defined outside the curly braces may still be part of the scope defined by the curly braces rather than the surrounding scope -- **function parameters** are a good example of this.

#### 4.10 Introduction to the 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.

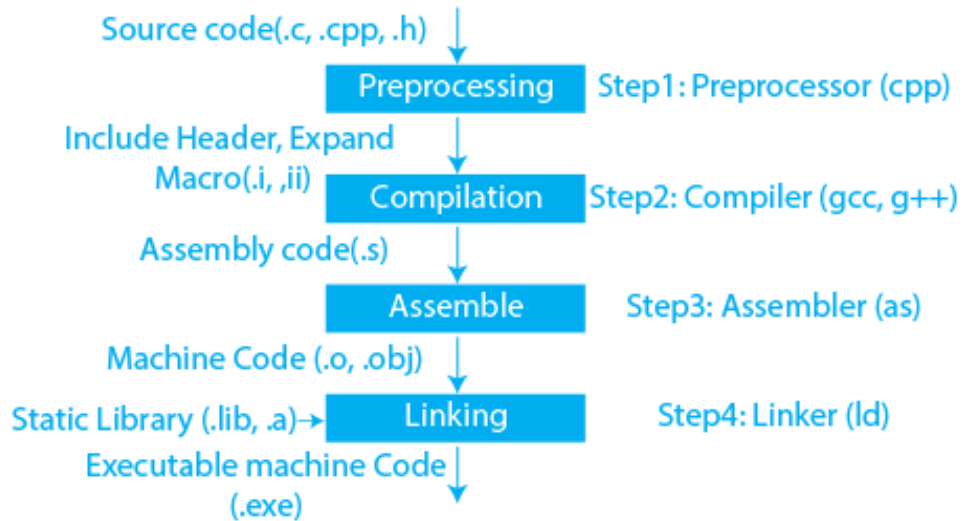
Historically, the preprocessor was a separate program from the compiler, **but in modern compilers, the preprocessor may be built right into the compiler itself.**

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.

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.

**The entire process** of preprocessing, compiling, and linking is called translation.

#### 4.10.1 Translation process



#### 4.10.2 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).

##### 4.10.2.1 Include

You've already seen the **#include** directive in action (generally to `#include <iostream>`). 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.

Once the preprocessor has finished processing the code file plus all of the #included content, the result is called a **translation unit**. The translation unit is what is sent to the compiler to be compiled.

#### 4.10.2.2 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. Their use is generally **considered unsafe**, and almost anything they can do can be done by a normal function.

**Object-like macros** can be defined in one of two ways:

- **#define IDENTIFIER**
  - Used for **conditional compilation**
  - **#if**
  - **#ifdef**
  - **#ifndef**
  - **#endif**
- **#define IDENTIFIER substitution\_text**

The top definition has no substitution text, whereas the bottom one does. Because **these are** preprocessor **directives (not statements)**, note that **neither form ends with a semicolon.**

The identifier for a macro uses the same naming rules as normal identifiers: they can use letters, numbers, and underscores, cannot start with a number, and should not start with an underscore. By convention, macro names are typically all upper-case, separated by underscores.

#### 4.11 Header files

As programs grow larger (and make use of more files), it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file. Wouldn't it be nice if you could **put all your forward declarations in one place and then import them when you need them?**

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. The primary purpose of a header file is to propagate declarations to code (.cpp) files.

When you **#include** a file, the **content** of the included file is **inserted at the point of inclusion**. This provides a useful way to pull in declarations from another file.

**Note: including definitions in a header file results in a violation of the one-definition rule.**

**Best practice:**

- 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
- 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 #included 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.



#### 4.11.1 *Standard library header files*

Consider the following program:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Hello, world!";
6 |     return 0;
7 | }
```

This program prints “Hello, world!” to the console using `std::cout`. However, **this program never provided a definition or declaration for `std::cout`**, so how does the compiler know what `std::cout` is?

The answer is that **`std::cout` has been forward declared in the “`iostream`” header file**. When we `#include <iostream>`, **we’re requesting that the preprocessor copy all of the content (including forward declarations for `std::cout`) from the file named “`iostream`” into the file doing the `#include`**.

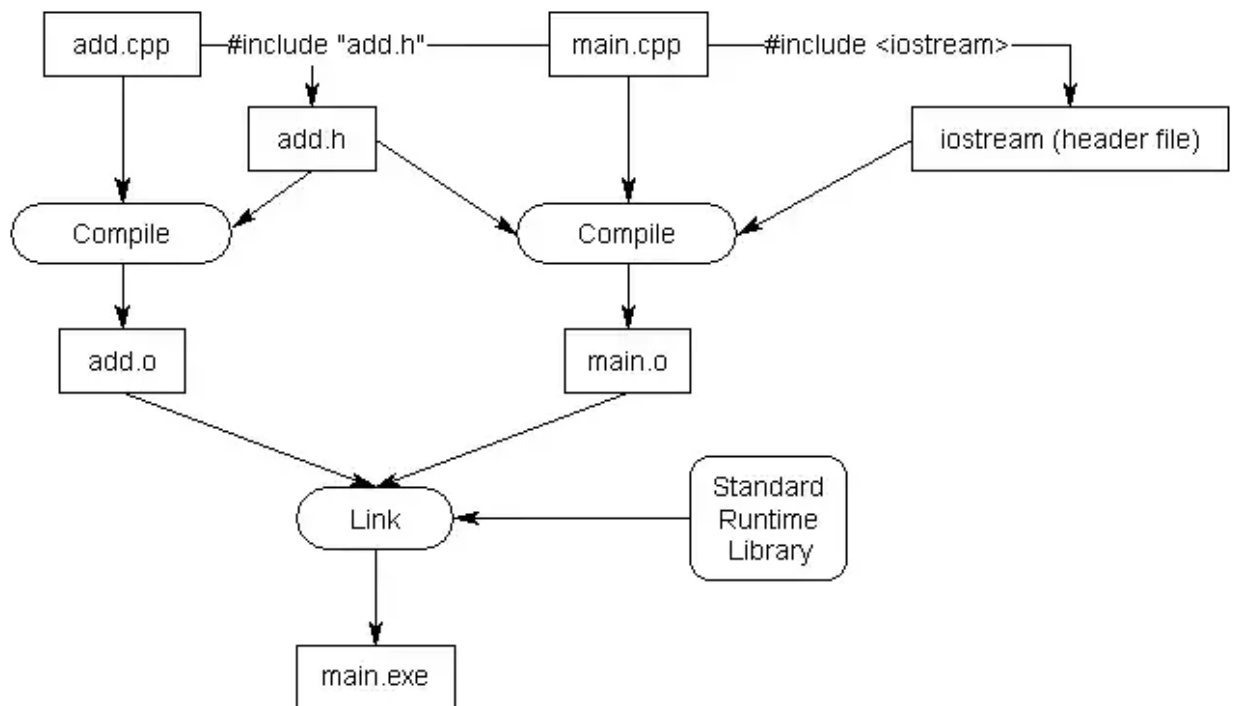
We use angled brackets here to help the preprocessor. This way we tell it that we didn’t write it ourselves and it will start looking in the **include directories**. These are configured according to the **IDE**, they can also be modified / configured according to the needs of a project.

#### 4.11.2 *Why doesn’t `iostream` have a `.h` extension?*

`iostream.h` is a different header file than `iostream`! **When C++ was first created, all of the headers in the standard library ended in a `.h` suffix**.

When the language was standardized by the ANSI committee, they decided to move all of the names used in the standard library into the **`std` namespace to help avoid naming conflicts** with user-declared identifiers. However, this presented a problem: if they moved all the names into the `std` namespace, none of the old programs (that included `iostream.h`) would work anymore!

To work around this issue, C++ introduced new header files that lack the `.h` extension. These new header files declare all names inside the `std` namespace. This way, older programs that include `#include <iostream.h>` do not need to be rewritten, and newer programs can `#include <iostream>`.



#### 4.11.3 *#include order*

To maximize the chance that missing includes will be flagged by compiler, order your `#includes` as follows:

1. The paired header file
2. Other headers from your project
3. 3rd party library headers
4. Standard library headers

The headers for each grouping should be sorted alphabetically (unless the documentation for a 3rd party library instructs you to do otherwise).

#### 4.11.4 Including header files from other directories

Another common question involves how to include header files from other directories.

One (**bad**) way to do this is to include a **relative path** to the header file you want to include as part of the `#include` line.

**For example:**

- `#include "headers/myHeader.h"`
- `#include "../moreHeaders/myOtherHeader.h"`

While this will compile (assuming the files exist in those relative directories), the downside of this approach is that **it requires you to reflect your directory structure in your code**. If you ever update your directory structure, your code won't work anymore.

A better method is to tell your compiler or IDE that you have a bunch of header files in some other location, so that it will look there when it can't find them in the current directory. This can generally be done by setting an include path or search directory in your IDE project settings.

The nice thing about this approach is that if you ever change your directory structure, you only have to change a single compiler or IDE setting instead of every code file.

#### 4.11.5 Headers may include other headers

It's common that a header file will need a declaration or definition that lives in a different header file. Because of this, **header files will often #include other header files**.

When your code file `#includes` the first header file, **you'll also get any other header files that the first header file includes** (and any header files those include, and so on). These additional header files are sometimes **called transitive includes**, as they're **included implicitly** rather than explicitly.

The content of these transitive includes are available for use in your code file. However, **you generally should not rely on the content of headers that are included transitively** (unless reference documentation indicates that those **transitive includes** are required). The implementation of header files may change over time, or be different across different systems. If that happens, your code may only compile on certain systems, or may compile now but not in the future. This is easily avoided by explicitly including all of the header files the content of your code file requires.

## 4.12 Header guards

### 4.12.1 *The duplicate definition problem*

In Forward declarations and definitions, we noted that a variable or function identifier can only have one definition (the one definition rule).

Thus, a program that defines a variable identifier more than once will cause a compile error. Similarly, programs that define a function more than once will also cause a compile error.

While these programs are easy to fix (remove the duplicate definition), 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).

square.h:

```
1 | int getSquareSides()
2 | {
3 |     return 4;
4 | }
```

wave.h:

```
1 | #include "square.h"
```

main.cpp:

```
1 | #include "square.h"
2 | #include "wave.h"
3 |
4 | int main()
5 | {
6 |     return 0;
7 | }
```

This seemingly innocent looking program won't compile! Here's what's happening. First, main.cpp #includes square.h, which copies the definition for function getSquareSides into main.cpp. Then main.cpp #includes wave.h, which #includes square.h itself. This copies contents of square.h (including the definition for function getSquareSides) into wave.h, which then gets copied into main.cpp.

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:

```
1 | #ifndef SOME_UNIQUE_NAME_HERE
2 | #define SOME_UNIQUE_NAME_HERE
3 |
4 | // your declarations (and certain types of definitions) here
5 |
6 | #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.** SOME\_UNIQUE\_NAME\_HERE can be any name you want, but **by convention is set to the full filename** of the header file, typed in all caps, using underscores for spaces or punctuation. For example, square.h would have the header guard:

```
1 | #ifndef SQUARE_H
2 | #define SQUARE_H
3 |
4 | int getSquareSides()
5 | {
6 |     return 4;
7 | }
8 |
9 | #endif
```

**Even the standard library headers use header guards.**

**SUMMARY:**

- Header guards **prevent a header file from being included multiple times within the same translation unit** (a single .cpp file and all the headers it includes).
- They **do not prevent the same header file from being included once in multiple translation units** (different .cpp files).
- **Duplicate declarations** are fine -- but even if your header file is composed of all declarations (no definitions) it's still a best practice to include header guards.

**BEST PRACTICE:**

This is the reason why we **shouldn't put functions in a header file**.

**4.12.2 *Can't we just avoid definitions in header files?***

We've generally told you not to include function definitions in your headers. So, you may be wondering why you should include header guards if they protect you from something you shouldn't do.

**There are quite a few cases we'll show you in the future where it's necessary to put non-function definitions in a header file.** For example, C++ will let you create your own types. These **custom types are typically defined in header files**, so the type definitions can be propagated out to the code files that need to use them. Without a header guard, a code file could end up with multiple (identical) copies of a given type definition, which the compiler will flag as an error.

So even though it's not strictly necessary to have header guards at this point in the tutorial series, **we're establishing good habits now, so you don't have to unlearn bad habits** later.

#### 4.12.3 *#pragma once*

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

**#pragma once** serves the same purpose as header guards: to avoid a header file from being included multiple times. With traditional header guards, the developer is responsible for guarding the header (by using preprocessor directives `#ifndef`, `#define`, and `#endif`). **With #pragma once, we're requesting that the compiler guard the header.** How exactly it does this is an implementation-specific detail.

**For most projects, #pragma once works fine,** and many developers now prefer it because it is easier and less error-prone. Many IDEs will also auto-include `#pragma once` at the top of a new header file generated through the IDE.

**Because #pragma once is not defined by the C++ standard, it is possible that some compilers may not implement it.** For this reason, **some development houses (such as Google) recommend using traditional header guards.** In this tutorial series, **we will favor header guards,** as **they are the most conventional way to guard headers.** However, support for `#pragma once` is fairly ubiquitous at this point, and **if you wish to use #pragma once instead, that is generally accepted in modern C++.**

## 4.13 How to design your first programs

### 4.13.1 Steps

1. Define your goal
  - In order to write a successful program, you first need to define what your goal is. Ideally, you should be able to state this in a sentence or two. It is often useful to express this as a user-facing outcome.
2. Define the requirements
  - While defining your problem helps you determine what outcome you want, it's still vague. The next step is to think about requirements.
  - Requirements is a fancy word for both the constraints that your solution needs to abide by (e.g. budget, timeline, space, memory, etc....), as well as the capabilities that the program must exhibit in order to meet the users' needs. Note that your requirements should similarly be focused on the "what", not the "how".
3. Define your tools, targets and backup plan
  - System architecture
  - Backup plans for your project
  - What parts of the standard library
  - Solo or co-op coding?
  - Define a testing / feedback / release strategy
4. Break hard problems down into easy problems
5. Figure out the sequence of events

### 4.13.2 Advice

- Keep your programs **simple to start**.
- **Add** features **over time**.
- **Focus on one area** at a time.
- **Test each piece of code** as you go.
- **Don't** invest in **perfecting early code**.
- **Optimize for maintainability**, not performance.



#### 4.14 Chapter 2 summary

A **function** is a reusable sequence of statements designed to do a particular job. Functions you write yourself are called **user-defined** functions.

A **function call** is an expression that tells the CPU to execute a function. The function initiating the function call is the **caller**, and the function being called is the **callee** or **called** function. Do not forget to include parenthesis when making a function call.

The curly braces and statements in a function definition are called the **function body**.

A function that returns a value is called a **value-returning function**. The **return type** of a function indicates the type of value that the function will return. The **return statement** determines the specific **return value** that is returned to the caller. A return value is copied from the function back to the caller -- this process is called **return by value**. Failure to return a value from a non-void function will result in undefined behavior.

The return value from function main is called a **status code**, and it tells the operating system (and any other programs that called yours) whether your program executed successfully or not. By consensus a return value of 0 means success, and a non-zero return value means failure.

Practice **DRY** programming -- “don’t repeat yourself”. Make use of variables and functions to remove redundant code.

Functions with a return type of **void** do not return a value to the caller. A function that does not return a value is called a void function or non-value returning function. Void functions can’t be called where a value is required.

A return statement that is not the last statement in a function is called an **early return**. Such a statement causes the function to return to the caller immediately.

A **function parameter** is a variable used in a function where the value is provided by the caller of the function. An **argument** is the specific value passed from the caller to the function. When an argument is copied into the parameter, this is called **pass by value**.

Function parameters and variables defined inside the function body are called **local variables**. The time in which a variable exists is called its **lifetime**. Variables are created and destroyed at **runtime**, which is when the program is running. A variable’s **scope** determines where it can be seen and used. When a variable can be seen and used, we say it is **in scope**. When it cannot be seen, it cannot be used, and we say it is **out of scope**. **Scope is a compile-time property**, meaning it is enforced at compile time.

**Whitespace** refers to characters used for formatting purposes. In C++, this includes spaces, tabs, and newlines.

A **forward declaration** allows us to tell the compiler about the existence of an identifier before actually defining the identifier. To write a forward declaration for a function, we use a **function prototype**, which includes the function's return type, name, and parameters, but no function body, followed by a semicolon.

A **definition** actually implements (for functions and types) or instantiates (for variables) an identifier. A **declaration** is a statement that tells the compiler about the existence of the identifier. In C++, all definitions serve as declarations. **Pure declarations** are declarations that are not also definitions (such as function prototypes).

Most non-trivial programs contain multiple files.

When two 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 error due to a **naming collision**. A **namespace** guarantees that all identifiers within the namespace are unique. The std namespace is one such namespace.

The **preprocessor** is a process that runs on the code before it is compiled. **Directives** are special instructions to the preprocessor. Directives start with a # symbol and end with a newline. A **macro** is a rule that defines how input text is converted to a replacement output text.

**Header files** are files designed to propagate declarations to code files. When using the #include directive, the #include directive is replaced by the contents of the included file. When including headers, use angled brackets when including system headers (e.g. those in the C++ standard library), and use double quotes when including user-defined headers (the ones you write). When including system headers, include the versions with no .h extension if they exist.

**Header guards** prevent the contents of a header from being included more than once into a given code file. They do not prevent the contents of a header from being included into multiple different code files.

## 5 Debugging

### 5.1 Syntax and semantic errors

A **syntax error** occurs when you write a statement that is **not valid according to the grammar** of the C++ language.

A **semantic error** occurs when a statement is **syntactically valid**, but **does not do what the programmer intended**.

### 5.2 Process

- Find the root cause of the problem
  - Find the line of code that's not working correctly
- Understand the problem
  - Why does it occur?
  - Should it even be fixed or is the behavior correct?
- Determine a fix
- Write the fix.
- Retest to ensure the problem has been fixed.
- Retest to ensure no new problems have emerged.

### 5.3 Strategy

1. Figure out how to reproduce the problem
2. Run the program and gather information to narrow down where the problem is
3. Repeat the prior step until you find the problem

#### 5.3.1 Tactics

1. Commenting out the code
2. Validate the code flow
3. Print out values
4. Use integrated debug mode
  - a. Breakpoints
  - b. Stepping
  - c. Watching variables
  - d. Watch the call stack

## 6 Fundamental data types

### 6.1 Introduction to fundamental data types

#### 6.1.1 *Bits, bytes, and memory addressing*

Computers have random access memory (RAM) that is available for programs to use. When a variable is defined, **a piece of that memory is set aside** for that variable.

The smallest unit of memory is a **binary digit** (also called a **bit**), which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch -- either the light is off (0), or it is on (1). There is no in-between. If you were to look at a random segment of memory, all you would see is ...011010100101010... or some combination thereof.

Memory is organized into sequential units called **memory addresses** (or **addresses** for short). Similar to how a street address can be used to find a given house on a street, the memory address allows us to find and access the contents of memory at a particular location.

Perhaps surprisingly, in modern computer architectures, each bit does not get its own unique memory address. This is because the number of memory addresses is limited, and the need to access data bit-by-bit is rare. Instead, each memory address holds 1 byte of data. A byte is a group of bits that are operated on as a unit. The modern standard is that a **byte** is comprised of **8 sequential bits**.

### 6.1.2 *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. You have already seen one example of a data type: the integer. When we declare a variable as an integer, we are telling the compiler “The piece of memory that this variable uses is going to be interpreted as an integer value”.

When you give an object a value, **the compiler and CPU** take care of **encoding** your value **into** the appropriate **sequence of bits** for that data type, **which are then stored in memory** (remember: memory can only store bits). For example, if you assign an integer object the value 65, that value is converted to the sequence of bits 0100 0001 and stored in the memory assigned to the object.

Conversely, when the object is evaluated to produce a value, that sequence of bits is reconstituted back into the original value. Meaning that 0100 0001 is converted back into the value 65.

Fortunately, the compiler and CPU do all the hard work here, so **you generally don’t need to worry about how values get converted into bit sequences and back.**

### 6.1.3 Fundamental data types

C++ comes with built-in support for many different data types. These are called **fundamental data types**, but are often informally called **basic types**, **primitive types**, or **built-in 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

### 6.1.4 The \_t suffix

Many of the types defined in newer versions of C++ (e.g. `std::nullptr_t`) use a `_t` suffix. **This suffix means “type”, and it’s a common nomenclature applied to modern types.**

If you see something with a `_t` suffix, it’s probably a type. But many types don’t have a `_t` suffix, so this isn’t consistently applied.

## 6.2 Void

**Void** is the easiest of the data types to explain. Basically, void means “**no type**”!

Void is our first example of an **incomplete type**.

An incomplete type is a type that has been declared but not yet defined. The compiler knows about the existence of such types, but does not have enough information to determine how much memory to allocate for objects of that type. void is intentionally incomplete since it represents the lack of a type, and thus cannot be defined.

Void is a bit of a special type and it's only used in certain contexts.

**For example:** a void type variable doesn't work. This is because it is an incomplete type and they **cannot be instantiated**.

### 6.2.1 Void contexts

- Most commonly, void is used to indicate that a **function does not return a value**.
  - Note: a **value-returning return statement will cause a compiler error**.
- Deprecated functions that don't take parameters (**C-specific**)
  - We put **void as a parameter** without an identifier.
  - Note: Although this will compile in C++ (for backwards compatibility reasons), **this use of keyword void is considered deprecated in C++**.
  - **Best practice:** Empty parameter list instead of void. (**implies an implicit void**)
- Void pointers
  - Will be discussed later on since it's an advanced use of the keyword.

## 6.3 Object sizes and the sizeof operator

### 6.3.1 Object sizes

To recap, memory on modern machines is typically organized into **byte-sized units**, with **each byte** of memory having a **unique address**.

Up to this point, it has been useful to think of memory as a bunch of cubbyholes or mailboxes where we can put and retrieve information, and variables as names for accessing those cubbyholes or mailboxes. However, **this analogy is not quite correct in one regard**.

**Most objects actually take up more than 1 byte of memory.** A single object may use 1, 2, 4, 8, or even more consecutive memory addresses. The amount of memory that an object uses is based on its data type.

Because we typically access memory through variable names (and not directly via memory addresses), the compiler is able to hide the details of how many bytes a given object uses from us. When we access some variable *x*, the compiler knows how many bytes of data to retrieve (based on the type of variable *x*), and can handle that task for us. **Even so, there are several reasons it is useful to know how much memory an object uses.**

First, **the more memory an object uses, the more information it can hold.**

A single bit can hold **2 possible values**, a 0, or a 1

2 bits can hold **4 possible values**

3 bits can hold **8 possible values**

To generalize, an object with *n* bits (where ***n* is an amount**) can hold  $2^n$  (2 to the power of *n*, also commonly written  **$2^n$** ) unique values.

Example 1:

**2-byte object (2 bytes = 8 bits)**

**Formula:  $2^n = 2^8 = 256$  possible values, range 0 - 255**

Example 2:

**4-byte integer (4 bytes = 32 bits)**

**Formula:  $2^n = 2^{32} = 4294967296$  possible values, range 0 - 4294967295**



Thus, the size of the object puts a limit on the amount of unique values it can store.

Second, **computers have a finite amount of free memory**. Every time we define an object, a small portion of that free memory is used for as long as the object is in existence. Because modern computers have a lot of memory, this impact is usually negligible. **However, for programs that need a large amount of objects or data (e.g. a game that is rendering millions of polygons), the difference between using 1-byte and 8-byte objects can be significant.**

### 6.3.2 Fundamental data type sizes

The obvious next question is “how much memory do variables of different data types take?”.

Surprisingly the **C++ standard does not define the exact size in bits**. However, it **does define a minimum size** (in bits).

In this tutorial series, we will take a simplified view, by making some reasonable assumptions that are generally true for modern architectures:

- **A byte is 8 bits.**
- Memory is byte addressable, so the **smallest object is 1 byte**
- Floating point support is IEEE-754 compliant.
- We are on a **32-bit or 64-bit architecture**.

Category	Type	Minimum Size	Typical Size	Note
Boolean	bool	1 byte	1 byte	
character	char	1 byte	1 byte	always 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	
integer	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	

Tip:

**For maximum portability, you shouldn't assume that variables are larger than the specified minimum size.**

Alternatively, if you want to assume that a type has a certain size (e.g. that an int is at least 4 bytes), you can use `static_assert` to have the compiler fail a build if it is compiled on an architecture where this assumption is not true. We cover how to do this in the Assert and `static_assert` lesson.

### 6.3.3 *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 its size in bytes.

You can also use the `sizeof` operator on a **variable name**.

`sizeof` **does not include dynamically allocated memory** used by an object. We discuss dynamic memory allocation in a future lesson.

## 6.4 Signed integers

An integer is an integral type that can represent positive and negative whole numbers, including 0. C++ has **4 primary fundamental integer types** available for use:

- **Short int**            **16 bits**
- **Int**                **16 – 32 bits**    (Typically, 32 bits on modern architectures)
- **Long int**           **32 bits**
- **Long long int**    **64 bits**

The key difference between the various integer types is that **they have varying sizes**.

**Technically**, the **bool** and **char** types **are considered to be integral types** (because **these types store their values as integer values**). For the purpose of the next few lessons, we'll exclude these types from our discussion.

When writing negative numbers in everyday life, we use a negative sign. For example, -3 means "negative 3". We'd also typically recognize +3 as "positive 3" (though common convention dictates that we typically omit plus prefixes).

**This attribute of being positive, negative, or zero is called the number's sign.**

By default, integers in C++ are **signed**, which means the number's **sign is stored as part of the number**. Therefore, a signed integer can hold both positive and negative numbers (and 0).

In binary representation, a single bit (called the **sign bit**) is used to store the sign of the number. The non-sign bits (called the **magnitude bits**) determine the magnitude of the number.

Best practice for defining:

Prefer the **shorthand types** that do not use the int suffix or signed prefix.

### 6.4.1 Signed integer ranges

As you learned in the last section, a variable with  $n$  bits can hold  $2^n$  possible values. But which specific values? We call the set of specific values that a data type can hold its **range**. The range of an integer variable is determined by **two factors**: its size (in bits), and whether it is signed or not.

By definition, an 8-bit signed integer has a range of -128 to 127. This means a signed integer can store any integer value between -128 and 127 **(inclusive) safely**.

Here's a table containing the range of signed integers of different sizes:

Size/Type	Range
8 bit signed	-128 to 127
16 bit signed	-32,768 to 32,767
32 bit signed	-2,147,483,648 to 2,147,483,647
64 bit signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

For the math inclined, an n-bit signed variable has a range of  $-(2^{n-1})$  to  $2^{n-1}$ .

#### 6.4.2 *Overflow*

What happens if we try to assign a value that is higher than the range of the fundamental data type?

The C++20 standard makes this blanket statement: "If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the **behavior is undefined**". Colloquially, this is called **overflow**.

If an arithmetic operation (such as addition or multiplication) attempts to create a value outside the range that can be represented, this is called **integer overflow** (or **arithmetic overflow**). For signed integers, integer overflow will result in undefined behavior.

In general, **overflow results in information being lost**, which is almost never desirable. If there is any suspicion that an object might need to store a value that falls outside its range, use a type with a bigger range!

#### 6.4.3 *Integer division*

When doing division with two integers (called **integer division**), C++ always produces an integer result. Since integers can't hold fractional values, any fractional portion is simply dropped (not rounded!).

Warning: Be careful when using integer division, as **you will lose any fractional parts of the quotient**. However, if it's what you want, integer division is safe to use, as the results are predictable.

#### 6.4.4 Two's compliment

Two's compliment representation is a way of representing positive and negative numbers in binary.

**Let's take 001000 for example:** first we calculate the indexes separately because they make part of the whole number. In binary if any number at a certain position is set to 1 then we add them to know what the whole byte evaluates to.

##### 6.4.4.1 Positive two's compliment

$$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ -2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \end{array}$$

In this case  $2^3$  is 8. Also note that the 0 with the most value is signed (negative)

##### 6.4.5 Negative two's compliment

The same rules as before apply except this time the signed bit is set to 1 and for this value it counts  $-2^5$  (-32). This is important because when we add all of the positive magnitude bits we then also subtract the signed bit.

$$\begin{array}{l} 101100 \\ = -2^5 + 2^3 + 2^2 \\ = -32 + 8 + 4 \\ = -20 \end{array}$$

## 6.5 Unsigned integers, and why to avoid them

We covered signed integers, which are a set of types that can hold positive and negative whole numbers, including 0.

C++ also supports unsigned integers. Unsigned integers are integers that can only hold non-negative whole numbers.

To define an unsigned integer, we **use the unsigned keyword**. By convention, this is placed before the type.

A 1-byte unsigned integer has a range of 0 to 255. Compare this to the 1-byte signed integer range of -128 to 127. **Both can store 256 different values, but signed integers use half of their range for negative numbers, whereas unsigned integers can store positive numbers that are twice as large.**

When no negative numbers are required, unsigned integers are well-suited for networking and systems with little memory, because unsigned integers can store more positive numbers without taking up extra memory.

### 6.5.1 Overflow

Important note: Oddly, the **C++ standard explicitly says “a computation involving unsigned operands can never overflow”**. This is contrary to general programming consensus that integer overflow encompasses both signed and unsigned use cases. **Given that most programmers would consider this overflow, we’ll call this overflow despite the C++ standard’s statements to the contrary.**

If an unsigned value is out of range, it is divided by one greater than the largest number of the type, and only the **remainder kept**.

### 6.5.2 Controversy over unsigned numbers

Many developers (and some large development houses, such as Google) believe that developers should generally avoid unsigned integers. This is largely **because of two behaviors** that can cause problems.

First, with signed values, it takes a little work to accidentally overflow the top or bottom of the range because those values are far from 0. **With unsigned numbers, it is much easier to overflow the bottom of the range, because the bottom of the range is 0, which is close to where the majority of our values are.** Another common **unwanted wrap-around happens when an unsigned integer is repeatedly decremented by 1**, until it tries to decrement to a negative number. You'll see an example of this when loops are introduced.

Second, and more insidiously, **unexpected behavior can result when you mix signed and unsigned integers.** In C++, if a mathematical operation (e.g. arithmetic or comparison) has one signed integer and one unsigned integer, the **signed integer will usually be converted to an unsigned integer. And the result will thus be unsigned.**

**Best practice:** Favor signed numbers over unsigned numbers for holding quantities (even quantities that should be non-negative) and mathematical operations. Avoid mixing signed and unsigned numbers.

### 6.5.3 So, when should you use unsigned numbers?

There are still a few cases in C++ where it's okay / necessary to use unsigned numbers.

First, unsigned numbers are preferred when dealing with **bit manipulation** (covered in chapter 0 -- that's a capital 'o', not a '0'). They are also useful when well-defined wrap-around behavior is required (**useful in some algorithms like encryption and random number generation**).

Second, use of unsigned numbers is still unavoidable in some cases, mainly those having to do with **array indexing**. We'll talk more about this in the lessons on arrays and array indexing.

Also note that if you're developing for an embedded system (e.g. an Arduino) or some other **processor/memory limited context**, use of unsigned numbers is more common and accepted (and in some cases, unavoidable) for performance reasons.

## 6.6 Fixed-width integers and `size_t`

we covered that C++ only guarantees that integer variables will have a minimum size -- but they could be larger, depending on the target system.

### 6.6.1 *Why isn't the size of the integer variables fixed?*

The short answer is that this goes back to the early days of C, when computers were slow and performance was of the utmost concern. C opted to intentionally leave the size of an integer open so that the compiler implementers could pick a size for `int` that performs best on the target computer architecture.

### 6.6.2 *Fixed-width integers*

To address the above issues, **C99 defined a set of fixed-width integers** (in the `stdint.h` header) that are guaranteed to be the same size on any architecture.

<code>std::int8_t</code>	1 byte signed	-128 to 127	Treated like a signed char on many systems.
<code>std::uint8_t</code>	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems.
<code>std::int16_t</code>	2 byte signed	-32,768 to 32,767	
<code>std::uint16_t</code>	2 byte unsigned	0 to 65,535	
<code>std::int32_t</code>	4 byte signed	-2,147,483,648 to 2,147,483,647	
<code>std::uint32_t</code>	4 byte unsigned	0 to 4,294,967,295	
<code>std::int64_t</code>	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
<code>std::uint64_t</code>	8 byte unsigned	0 to 18,446,744,073,709,551,615	



### 6.6.3 *Fixed-width integers downsides*

The fixed-width integers have **two downsides** that are typically raised.

First, the fixed-width integers are **not guaranteed to be defined on all architectures**. They **only exist on systems where there are fundamental types matching their widths** and following a certain binary representation. Your program will fail to compile on any such architecture that does not support a fixed-width integer that your program is using. However, given that most modern architectures have standardized around 8/16/32/64-bit variables, this is unlikely to be a problem unless your program needs to be portable to some exotic mainframe or embedded architectures.

Second, if you use a fixed-width integer, **it may be slower than a wider type on some architectures**. For example, if you need an integer that is guaranteed to be 32-bits, you might decide to use `std::int32_t`, but your CPU might actually be faster at processing 64-bit integers. However, just because your CPU can process a given type faster doesn't mean your program will be faster overall -- **modern programs are often constrained by memory usage rather than CPU, and the larger memory footprint may slow your program more than the faster CPU processing accelerates it. It's hard to know without actually measuring.**

### 6.6.4 *std::int8\_t and std::uint8\_t likely behave like chars instead of integers*

**Due to an oversight in the C++ specification**, most compilers define and treat `std::int8_t` and `std::uint8_t` (and the corresponding fast and least fixed-width types) identically to types signed char and unsigned char respectively. This means these 8-bit types may (or may not) behave differently than the rest of the fixed-width types, which can lead to errors. **This behavior is system-dependent, so a program that behaves correctly on one architecture may not compile or behave correctly on another architecture.**

### 6.6.5 *Fast and least integers*

To help address the above downsides, **C++ also defines two alternative sets of integers that are guaranteed to be defined.**

The **fast types** (`std::int_fast#_t` and `std::uint_fast#_t`) provide the fastest signed/unsigned integer type with a width of at least # bits (**where # = 8, 16, 32, or 64**). **For example**, `std::int_fast32_t` will give you the fastest signed integer type that's at least 32 bits. **By fastest, we mean the integral type that can be processed most quickly by the CPU.**

The **least types** (`std::int_least#_t` and `std::uint_least#_t`) provide the smallest signed/unsigned integer type with a width of at least # bits (**where # = 8, 16, 32, or 64**). **For example**, `std::uint_least32_t` will give you the **smallest** unsigned integer type that's at least 32 bits.

However, these **fast and least integers have their own downsides**: First, not many programmers actually use them, and a **lack of familiarity can lead to errors**. Second, the **fast types can lead to memory wastage, as their actual size may be larger than indicated by their name**. More seriously, because the **size of the fast/least integers can vary**, it's possible that your program may exhibit different behaviors on architectures where they resolve to different sizes.

#### **Summary:**

- **Fixed-width integers** aimed to standardize integer sizes across platforms but **aren't universally supported** and can be **suboptimal in performance on some architectures**.
- **Fast and least integers** were introduced to provide more flexible options that adapt to the target architecture's strengths but come with their own trade-offs, like potential **memory inefficiency (fast) and varying sizes (least)**.
- The **8-bit fixed-width, fast and least integer types** are **often treated like chars instead of integer values** (and this may vary per system). Prefer the 16-bit fixed integral types for most cases.

### 6.6.6 Best practices

- Prefer **int** when the size of the integer doesn't matter (e.g. the number will always fit within the range of a 2-byte signed integer) and the variable is short-lived (e.g. destroyed at the end of the function). For example, if you're asking the user to enter their age, or counting from 1 to 10, it doesn't matter whether int is 16 or 32 bits (the numbers will fit either way). **This will cover the vast majority of the cases you're likely to run across.**
- Prefer **std::int#\_t** when storing a quantity that needs a guaranteed range.
- Prefer **std::uint#\_t** when doing bit manipulation or where well-defined wrap-around behavior is required.

**Avoid** the following when possible:

- short and long integers, **use a fixed-width type** instead.
- **Unsigned types for holding quantities.**
- The **8-bit fixed-width integer types.**
- The fast and least fixed-width types.
- Any compiler-specific fixed-width integers
  - For example, Visual Studio defines `__int8`, `__int16`, etc...

### 6.6.7 Std::size\_t

The **sizeof** operator **returns** a value of type **std::size\_t**. `std::size_t` is an alias for an **implementation-defined unsigned integral type**.

In other words, **the compiler decides** if `std::size_t` is an unsigned int, an unsigned long, an unsigned long long, etc...

**For advanced readers:** `std::size_t` is actually a typedef.

**Best practice:** If you use `std::size_t` explicitly in your code, **#include** one of the **headers that defines `std::size_t`** (we recommend `<cstdint>`).

**Using `sizeof` does not require a header** (even though it return a value whose type is `std::size_t`).

## 6.7 Introduction to scientific notation

**Scientific notation** is a useful shorthand for writing lengthy numbers in a concise manner. And although scientific notation may seem foreign at first, understanding scientific notation will help you understand how floating-point numbers work, and more importantly, what their limitations are.

Numbers in scientific notation take the following form: ***significand***  $\times 10^{\text{exponent}}$

By convention, numbers in scientific notation are written with one digit before the decimal point, and the rest of the digits afterward.

Consider the mass of the Earth. In decimal notation, we'd write this as **597220000000000000000000 kg**. That's a really large number (too big to fit even in an 8-byte integer). It's also hard to read (is that 19 or 20 zeros?). Even with separators (5,972,200,000,000,000,000,000,000) the number is still hard to read.

In scientific notation, this would be written as  **$5.9722 \times 10^{24}$  kg**, which is **much easier to read**. Scientific notation has the added benefit of making it easier to compare the magnitude of two extremely large or small numbers simply by comparing the exponent.

Because it can be hard to type or display exponents in C++, we use the letter 'e' (or sometimes 'E') to represent the "**times 10 to the power of**" part of the equation. For example,  **$1.2 \times 10^4$  would be written as 1.2e4**, and  $5.9722 \times 10^{24}$  would be written as 5.9722e24.

### 6.7.1 Significant digits

Let's say you need to know the value of the mathematical constant **pi** for some equation, but you've forgotten. You ask two people. One tells you the value of pi is **3.14**. The other tells you the value of pi is **3.14159**. Both of these values are "correct", but **the latter is far more precise**.

Here's the most important thing to understand about scientific notation: The digits in the significand (the part before the 'e') are called the **significant digits** (or **significant figures**). The more significant digits, the more precise a number is.

In scientific notation, we'd write 3.14 as 3.14e0. Since there are **3 numbers** in the significand, this number **has 3 significant digits**. 3.14159 would be written as 3.14159e0. Since there are 6 numbers in the significand, this number has 6 significant digits.

### 6.7.2 How to convert decimal numbers to scientific notation

Use the following procedure:

- Your **exponent starts at zero**.
- **Slide** the decimal **left** or **right** so **there is only one non-zero digit** to the left of the decimal.
  - Each place you **slide** the decimal to the **left** **increases the exponent by 1**.
  - Each place you **slide** the decimal to the **right** **decreases the exponent by 1**.
- **Trim** off any leading **zeros** (**on the left** end of the significand)
- **Trim** off any trailing **zeros** (**on the right** end of the significand) only **if the original number had no decimal point**. We're assuming they're not significant. If you have additional information to suggest they are significant, you can keep them.

## 6.8 Floating point numbers

Integers are great for counting whole numbers, but sometimes we need to store very large (positive or negative) numbers, or **numbers with a fractional component**. A **floating-point type** variable is a variable that can hold a number with a fractional component, such as 4320.0, -3.33, or 0.01226. The floating part of the name floating point refers to the fact that the decimal point can **"float"** -- that is, it can support a variable number of digits before and after the decimal point.

When writing floating point numbers in your code, the decimal separator must be a decimal point. If you're from a country that uses a decimal comma, you'll need to get used to using a decimal point instead.

### 6.8.1 Floating point data types

There are **three standard floating point data types**: a **single-precision float**, a **double-precision double**, and an **extended-precision long double**. As with integers, C++ does not define the actual size of these types.

Floating point data types are **always signed** (can hold positive and negative values).

<b>Float</b>	<b>4 bytes</b>
<b>double</b>	<b>8 bytes</b>
<b>long double</b>	<b>8, 12, or 16 bytes</b>

On modern architectures, floating point representation for **float and double almost always follows IEEE 754 binary format** (created by William Kahan). In this format, **a float is 4 bytes and a double is 8 bytes**.

**Long double is a strange one.** Depending on the implementation, it may be equivalent to an IEEE 754 double (8 bytes), **it may be a 16-byte IEEE 754 binary format value, or it may be an 80-bit (typically padded to 12 bytes) non-IEEE 754 binary format value.** **We recommend avoiding long double.**

### 6.8.2 Best practice

Always make sure the type of your literals match the type of the variables they're being assigned to or used to initialize. Otherwise, an unnecessary conversion will result, possibly with a loss of precision.

e.g.:

**double pi { 3.14 };      no suffix means double type by default**

**float    pi { 3.14f };**

### 6.8.3 Floating point range

Assuming IEEE 754 representation for 4-, 8-, and 16-byte representations:

Size	Range	Precision
4 bytes	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ and 0.0	6-9 significant digits, typically 7
8 bytes	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$ and 0.0	15-18 significant digits, typically 16
80-bits (typically uses 12 or 16 bytes)	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$ and 0.0	18-21 significant digits
16 bytes	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$ and 0.0	33-36 significant digits

#### 6.8.4 *Floating point precision*

Consider the fraction  $1/3$ . The decimal representation of this number is 0.333333333333... with **3's going out to infinity**. If you were writing this number on a piece of paper, your arm would get tired at some point, and you'd eventually stop writing. And the number you were left with would be close to 0.3333333333... (with 3's going out to infinity) but not exactly.

On a computer, an **infinite precision number would require infinite memory to store**, and **we typically only have 4 or 8 bytes per value**. This limited memory means floating point numbers can only store a certain number of significant digits -- any additional significant digits are either lost or represented imprecisely. The number that is actually stored may be close to the desired number, but not exact.

The **precision** of a floating-point type **defines how many significant digits** it can represent **without information loss**.

The number of digits of precision a floating-point type has depends on both the size (**floats have less precision than doubles**) and the particular value being stored (some values can be represented more precisely than others).

For example, a float has 6 to 9 digits of precision. This means that **a float can exactly represent any number with up to 6 significant digits**. A number with **7 to 9** significant digits **may or may not be represented exactly** depending on the specific value. And a number with more than 9 digits of precision will definitely not be represented exactly.

**Double** values have between **15 and 18 digits of precision**, with most double values having at least 16 significant digits. **Long double** has a minimum precision of **15, 18, or 33 significant digits** depending on how many bytes it occupies.

### 6.8.5 *Outputting floating point values*

When outputting floating point numbers, **std::cout** has a **default precision of 6**.

that is, it **assumes** all **floating-point variables are only significant to 6 digits** (the minimum precision of a float), and hence **it will truncate anything after that**.

Also **note that std::cout will switch to outputting** numbers in **scientific notation in some cases**. Depending on the compiler, the exponent will typically be padded to a minimum number of digits.

We can **override** the **default precision that std::cout shows** by using an output manipulator function named **std::setprecision()**. **Output manipulators** alter how data is output, and are defined in the **iomanip** header.

Output manipulators (and input manipulators) are **sticky** -- meaning if you set them, they will remain set. The **one exception is std::setw**. Some IO operations reset **std::setw**, so **std::setw** should be used every time it is needed.

**Best practice:**

**Favor double** over float **unless space is at a premium**, as the lack of precision in a float will often lead to inaccuracies.

Also remember that **rounding errors make** floating point **comparisons tricky**. E.g.: Comparing 0.1 to 0.10000001 will be tricky and if you want this to evaluate to true you'd need to implement your own comparison logic or reduce precision on the comparison.

### 6.8.6 *NaN and Inf*

There are **two special categories** of floating-point numbers. These special categories are tied to **zero division**. The first is **Inf**, which represents **infinity**. **Inf can be positive or negative**.

The second is **NaN**, which stands for "**Not a Number**". There are several different kinds of NaN (which we won't discuss here). NaN and Inf are only available if the compiler uses a specific format (IEEE 754) for floating point numbers.

Inf stands for infinity, and **Ind** stands for **indeterminate**. Note that the **results of printing Inf and NaN are platform specific**, so your results may vary.

Best practice: **Avoid division by 0.0 altogether**, even if your compiler supports it.



## 6.9 Boolean values

In real-life, it's common to ask or be asked questions that can be answered with "yes" or "no". "Is an apple a fruit?" Yes. "Do you like asparagus?" No.

Now consider a similar statement that can be answered with a **"true" or "false"**: "Apples are a fruit". It's clearly true. Or how about, "I like asparagus". Absolutely false (yuck!).

These kinds of sentences that have only two possible outcomes: yes/true, or no/false are so common, that many programming languages include a **special type for dealing with them**. That type is called a **Boolean type** (note: Boolean is properly capitalized in the English language because it's named after its inventor, George Boole).

## 6.10 Boolean variables

Boolean variables are variables that can have only two possible values: **true**, and **false**.

To declare a Boolean variable, we use the keyword **bool**. E.g.: **bool b1 { true };**

When we print Boolean values, `std::cout` prints **0 for false**, and **1 for true**.

If you want `std::cout` to print true or false instead of 0 or 1, you can use **`std::boolalpha`**.

You can use **`std::noboolalpha`** to turn it back off.

## 6.11 Integer to Boolean conversion

When using uniform initialization, you can initialize a variable using **integer literals 0 (for false) and 1 (for true)** (but **you really should be using false and true** instead). **Other integer literals cause compilation errors.**

### 6.11.1 *Inputting Boolean values*

**Inputting Boolean values using `std::cin`** sometimes trips new programmers up.

It turns out that **`std::cin` only accepts two inputs for Boolean variables: 0 and 1** (not true or false). Any other inputs will cause `std::cin` to silently fail.

However, when **`std::boolalpha`** is enabled, **0 and 1 will no longer be interpreted as Booleans inputs (they both resolve to false as does any non-true input).**

Warning: Enabling `std::boolalpha` will **only allow lower-cased false or true to be accepted**. Variations with capital letters will not be accepted.

### 6.11.2 *Boolean return values*

Boolean values are often used as the return values for functions **that check whether something is true or not** (often called validation). Such functions are typically named starting with the word **is** (e.g. `isEqual`) or **has** (e.g. `hasCommonDivisor`).

## 6.12 Introduction to if statements

Consider a case where you want to do something only **IF** a certain condition is true.

Such conditions are common in programming, as they allow us to implement conditional behavior into our programs. The simplest kind of conditional statement in C++ is called an **if statement**. An if statement allows us to execute one (or more) lines of code only if some condition is true.

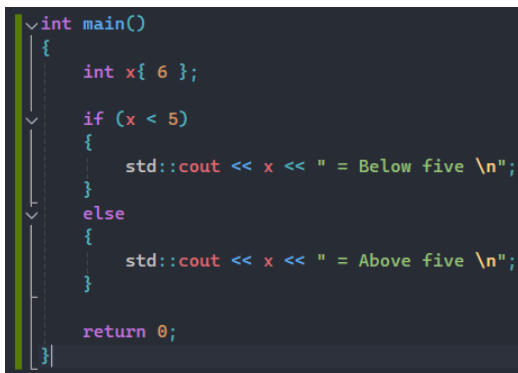
A **condition** (also called a **conditional expression**) is an expression that evaluates to a Boolean value.

**Syntax:**

```
if (health == 0)
{
    Respawn();
}
```

If the condition of an if statement evaluates to Boolean value true, then true\_statement is executed. **If** the condition instead evaluates to Boolean value **false**, then **true\_statement** is **skipped**.

### 6.12.1 If else example



```
int main()
{
    int x{ 6 };

    if (x < 5)
    {
        std::cout << x << " = Below five \n";
    }
    else
    {
        std::cout << x << " = Above five \n";
    }

    return 0;
}
```

### 6.12.2 Chaining if statements

Sometimes we want to check if several things are true or false in sequence. We can do so by chaining an if-statement (or if-else) to a prior if-else, like so:

```
int main()
{
    int x{ 5 };

    if (x == 5)
    {
        std::cout << x << " = Equal to five \n";
    }
    else if (x < 5)
    {
        std::cout << x << " = Below five \n";
    }
    else
    {
        std::cout << x << " = Above five \n";
    }

    return 0;
}
```

### 6.12.3 Boolean return values and if statements

Like we said in the last chapter on boolean values and **validation trough control-flow statements** like if statements go hand in hand.

```
bool IsEqual(int x)
{
    return x == 5;
}

bool IsBelow(int x)
{
    return x < 5;
}

int main()
{
    int x{ 6 };

    if (IsEqual(x))
    {
        std::cout << x << " = Equal to five \n";
    }
    else if (IsBelow(x))
    {
        std::cout << x << " = Below five \n";
    }
    else
    {
        std::cout << x << " = Above five \n";
    }

    return 0;
}
```

#### 6.12.4 Non-Boolean conditionals

In all of the examples above, our conditionals have been either Boolean values (true or false), Boolean variables, or functions that return a Boolean value. What happens if your conditional is an expression that does not evaluate to a Boolean value?

In such a case, **the conditional expression is converted to a Boolean value:**

- **non-zero** values get **converted to Boolean true (also negative values!)**
- **zero-values** get **converted to Boolean false.**

#### 6.12.5 If-statements and early returns

A return statement that is not the last statement in a function is called an **early return**. Such a **statement will cause the function to return to the caller** when the return statement is executed (**before the function would otherwise return to the caller**, hence, “early”). Make sure early return statements have a condition attached to them. **Unconditional early returns aren’t useful.**

```
bool IsBelowFive(int x)
{
    if (x < 5)
    {
        return true;
    }

    return false;
}
```

### 6.13 Chars

To this point, the fundamental data types we've looked at have been used to hold numbers (integers and floating points) or true/false values (Booleans). But what if we want to store letters or punctuation?

The **char** data type was designed to hold **a single character**. A character can be a single letter, **number**, **symbol**, or **whitespace**.

The char data type is an **integral type (which is still a fundamental type)**, meaning **the underlying value is stored as an integer**. Similar to how a Boolean value **0** is interpreted as **false** and non-zero is interpreted as **true**, **the integer stored by a char variable are interpreted as an ASCII character**.

ASCII stands for American Standard Code for Information Interchange, and it defines a particular way to represent English characters (plus a few other symbols) as **numbers between 0 and 127** (called an ASCII code or code point). For example, ASCII code 97 is interpreted as the character 'a'.

# ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

## 6.13.1 Initializing chars

You can initialize char variables using character literals or integer values:

```
char ch2{ 'a' };
```

```
char ch1{ 97 };
```

Warning: **Be careful not to mix up character numbers** with integer numbers. The following two initializations are not the same:

```
1 char ch{5}; // initialize with integer 5 (stored as integer 5)
2 char ch{'5'}; // initialize with code point for '5' (stored as integer 53)
```

### 6.13.2 Char size, range and default sign

Char is defined by C++ to always be **1 byte in size**. By default, **a char may be signed or unsigned** (though it's **usually signed**). **If you're using chars to hold ASCII characters, you don't need to specify a sign** (since both signed and unsigned chars can hold values between **0 and 127**).

**If you're using a char to hold small integers (something you should not do unless you're explicitly optimizing for space), you should always specify whether it is signed or unsigned.** A **signed char can hold a number between -128 and 127**. An unsigned char can hold a number between **0 and 255**.

### 6.13.3 Escape sequences

There are some sequences of characters in C++ that have special meaning. These characters are called **escape sequences**. An escape sequence **starts with a '\ (backslash)** character, and then a following letter or number.

**Warning:** Escape sequences start with a backslash (\), not a forward slash (/). If you use a forward slash by accident, it may still compile, but will not yield the desired result.

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash.
Question mark	\?	Prints a question mark. No longer relevant. You can use question marks unescaped.
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number



#### 6.13.4 *What's the difference between putting symbols in single and double quotes?*

Single **chars** are always put in **single quotes** (e.g. 'a', '+', '5'). A char can only represent one symbol (e.g. the letter a, the plus symbol, the number 5). **Text between double quotes** (e.g. "Hello, world!") is treated as a **string** of multiple characters.

Best practice: **Put stand-alone chars in single quotes** (e.g. 't' or '\n', **not "t" or "\n"**). This **helps the compiler optimize** more effectively.

#### 6.13.5 *Avoid multicharacter literals*

For backwards compatibility reasons, many **C++ compilers support multicharacter literals**, which are **char literals that contain multiple characters** (e.g. '56'). If supported, these have an **implementation-defined** value (meaning it varies depending on the compiler). Because they are not part of the C++ standard, and their value is not strictly defined, **multicharacter literals should be avoided**.

#### 6.13.6 *What about the other char types, wchar\_t, char8\_t, char16\_t, and char32\_t?*

**wchar\_t should be avoided in almost all cases (except when interfacing with the Windows API)**. Its size is implementation defined, and is not reliable. It has largely been **deprecated**.

Much like ASCII maps the integers 0-127 to American English characters, other character encoding standards exist to map integers (of varying sizes) to characters in other languages. The most well-known mapping outside of ASCII is the **Unicode standard**, which maps over 144,000 integers to characters in many different languages. Because Unicode contains so many code points, a single Unicode code point needs 32-bits to represent a character (called UTF-32). However, Unicode characters can also be encoded using multiple 16-bit or 8-bit characters (**called UTF-16 and UTF-8 respectively**).

**char16\_t and char32\_t were added to C++11 to provide explicit support for 16-bit and 32-bit Unicode characters**. These char types have the same size as std::uint\_least16\_t and std::uint\_least32\_t respectively (but are distinct types). **char8\_t was added in C++20 to provide support for 8-bit Unicode (UTF-8)**. It is a distinct type that uses the same representation as unsigned char.

You won't need to use char8\_t, char16\_t, or char32\_t unless you're planning on making your program Unicode compatible. Unicode and localization are generally outside the scope of these tutorials, so we won't cover it further.

## 6.14 Introduction to type conversion and static\_cast

Consider the following program:

```
1  #include <iostream>
2
3  void print(double x) // print takes a double parameter
4  {
5      std::cout << x << '\n';
6  }
7
8  int main()
9  {
10     print(5); // what happens when we pass an int value?
11
12     return 0;
13 }
```

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 a value 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, resulting in the following output: 5.

### 6.14.1 Type conversion produces a new value

Even though it is called a conversion, a **type conversion does not actually change the value or type** of the value being converted. Instead, the value to be converted is used as input, and the conversion results in a new value of the target type (via direct initialization).

### 6.14.2 *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**.

Assume we're passing a **floating-point type to an integer type**. 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.

Because converting a floating-point value to an integral value results in any fractional component being dropped, the compiler will warn us when it does an implicit type conversion from a floating point to an integral value. This happens even if we were to pass in a floating-point value with no fractional component, like 5.0 -- no actual loss of value occurs during the conversion to integral value 5 in this specific case, but the compiler may still warn us that the conversion is unsafe.

### 6.14.3 *An introduction to explicit type conversion via the static\_cast operator*

What if we intentionally wanted to pass a double value to a function taking an integer (knowing that the converted value would drop any fractional component?) **Turning off "treat warnings as errors" just to make our program compile is a bad idea**, because then we'll have warnings every time we compile (which we will quickly learn to ignore), and we **risk overlooking warnings about more serious issues**.

**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**. The syntax for the static cast looks a little funny:

```
static_cast<new_type>(expression)
```

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).

**Key insight:** Whenever you see C++ syntax (**excluding** the **preprocessor**) that makes use of angled brackets (<>), the thing **between the angled brackets will most likely be a type**. This is typically how C++ deals with code that need a parameterized type.

```
4  void print(int x)
5  {
6      std::cout << x << '\n';
7  }
8
9  int main()
10 {
11     print(static_cast<int>(5.5)); // explicitly convert double value 5.5 to an int
12
13     return 0;
14 }
```

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).

C++ supports other types of casts. We talk more about the different types of casts in the future.

There are some other special reasons to use the static\_cast operator:

- Converting **char** to **int**
- Converting **unsigned** numbers to **signed** numbers
- std::int8\_t and std::uint8\_t likely **behave like chars** instead of integers

## 6.15 Fundamental data types summary

The smallest unit of memory is a **binary digit**, also called a **bit**. The smallest unit amount of memory that can be addressed (accessed) directly is a byte. The **modern standard** is that a **byte equals 8 bits**.

A **data type** tells the compiler how to interpret the contents of memory in some meaningful way.

C++ comes with support for many fundamental data types, including **floating point numbers, integers, boolean, chars, null pointers, and void**.

**Void** is used to indicate **no type**. It is **primarily used to indicate that a function does not return a value**.

**Different types** take **different amounts of memory**, and the **amount of memory** used **may vary by machine**.

The **sizeof** operator can be **used to return the size of a type in bytes**.

**Signed integers** are used for holding positive and negative whole numbers, including 0. The set of values that a specific data type can hold is called its range. When using integers, keep an eye out for overflow and integer division problems.

**Unsigned integers** only hold positive numbers (and 0), and should generally be avoided unless you're doing bit-level manipulation.

**Fixed-width integers** are integers with guaranteed sizes, but they may not exist on all architectures. The **fast and least integers** are the fastest and smallest integers that are at least some size. `std::int8_t` and `std::uint8_t` should generally be avoided, as they tend to behave like chars instead of integers.

**size\_t** is an unsigned integral type that is used to **represent the size or length of objects**.

**Scientific notation** is a **shorthand way of writing lengthy numbers**. C++ supports scientific notation in conjunction with floating point numbers. The digits in the significand (the part before the e) are called the significant digits.

**Floating point** is a set of types designed to hold real numbers (including those with a fractional component). The precision of a number defines how many significant digits it can represent without information loss. A **rounding error** can occur when too many significant digits are stored in a floating-point number that can't hold that much precision.

Rounding errors happen all the time, even with simple numbers such as 0.1. Because of this, you shouldn't compare floating point numbers directly.

The **Boolean** type is used to store a true or false value.

**If statements** allow us to execute one or more lines of code if some condition is true. The **conditional expression** of an if-statement is interpreted as a boolean value. An **else statement** can be used to execute a statement when a prior if-statement condition evaluates to false.

**Char** is used to store values that are interpreted as an ASCII character. When using chars, be careful not to mix up ASCII code values and numbers. Printing a char as an integer value requires use of `static_cast`.

Angled brackets are typically used in C++ to represent something that needs a **parameterizable type**. This is used with `static_cast` to determine what data type the argument should be converted to (e.g. `static_cast<int>(x)` will return the value of x as an int).

## 7 Constants and Strings

### 7.1 Constant variables (named 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** are constant values that **are associated with an identifier**. These are also sometimes called symbolic constants, or occasionally just constants.
- **Literal constants** are constant values that **are not associated with an identifier**.

Types of named constants:

- Constant **variables**.
- **Object-like macros with substitution text** (introduced in: Introduction to the preprocessor, with additional coverage in this lesson).
- **Enumerated** constants (covered in lesson 13.2 -- Unscoped enumerations).

So far, all of the variables we've seen have been non-constant -- **that is, their values can be changed at any time** (typically **by assigning a new value**).

However, there are many cases where it is useful to define variables with values that cannot be changed. For example, consider the gravity of Earth (near the surface): 9.8 meters/second<sup>2</sup>. This isn't likely to change any time. **Defining this value as a constant helps ensure that this value isn't accidentally changed**. Constants also have other benefits that we'll explore in subsequent lessons.

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

```
const double gravity {9.8};
```

### 7.1.1 *Const placing*

Although C++ will accept the const qualifier **either before or after the type**, it's much **more common to use const before the type** because it better follows standard English language convention where modifiers come before the object being modified (e.g. a "a green ball", not a "a ball green").

Due to the way that the compiler parses more complex declarations, some developers prefer placing the const after the type (because it is slightly more consistent). **This style is called "east const"**. While this style has some advocates (and some reasonable points), it **has not caught on significantly**.

**Best practice:** Place const before the type (because it is more conventional to do so).

### 7.1.2 *Const variables must be initialized*

Const variables **must be initialized** when you define them, and then that value cannot be changed via assignment.

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

### 7.1.3 *Naming conventions*

There are a number of different naming conventions that are used for const variables.

- C-style:
  - Programmers who have transitioned from C often prefer **underscored, upper-case names for const variables** (e.g. EARTH\_GRAVITY).
- C++-style:
  - More common in C++ is to use **intercapped names with a 'k' prefix** (e.g. kEarthGravity).

However, because const variables act like normal variables (except they cannot 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).



#### 7.1.4 *Const function parameters*

**Function parameters can be made constants** via the `const` keyword.

Note: We do not need to provide an explicit initializer for our `const` parameter. The value of the argument in the function call will be used as the initializer for `const` parameters.

Making a function parameter constant **enlists the compiler's help to ensure that the parameter's value is not changed inside the function**. However, in modern C++ we don't make value parameters `const` because we generally don't care if the function changes the value of the parameter (since **it's just a copy that will be destroyed at the end of the function** anyway). The `const` keyword also adds a small amount of unnecessary clutter to the function prototype.

Best practice: **Don't** use `const` when **passing by value**. Later we'll talk about two other ways to pass arguments to functions: **pass by reference**, and **pass by address**. When using either of these methods, **proper use of `const` is important**.

#### 7.1.5 *Const return values*

A function's **return value** may also be made `const`.

**For fundamental types, the `const` qualifier on a return type is simply ignored** (your compiler may generate a warning).

For other types (which we'll cover later), there is typically little point in returning `const` objects by value, because they are temporary copies that will be destroyed anyway. Returning a `const` value **can also impede certain kinds of compiler optimizations** (involving move semantics), which **can result in lower performance**.

Best practice: **Don't** use `const` when **returning by value**.

### 7.1.6 Object-like macros with substitution text

In: Introduction to the preprocessor, we discussed object-like macros with substitution text.

```
1 #include <iostream>
2
3 #define MY_NAME "Alex"
4
5 int main()
6 {
7     std::cout << "My name is: " << MY_NAME << '\n';
8
9     return 0;
10 }
```

When the preprocessor processes the file containing this code, it will replace `MY_NAME` (on line 7) with `"Alex"`. Note that `MY_NAME` is a name, and the substitution text is a constant value, so **object-like macros with substitution text are also named constants**.

### 7.1.7 Prefer constant variables to preprocessor macros

So why not use preprocessor macros for named constants? **There are (at least) three major problems.**

The biggest issue is that **macros don't follow normal C++ scoping rules**. Once a macro is `#defined`, all subsequent occurrences of the macro's name in the current file will be replaced. If that name is used elsewhere, you'll get macro substitution where you didn't want it. This will most likely lead to strange compilation errors.

Second, it is often **harder to debug code using macros**. Although your source code will have the macro's name, the compiler and debugger never see the macro because it has already been replaced before they run. **Many debuggers are unable to inspect a macro's value**, and often have limited capabilities when working with macros.

Third, **macro substitution behaves differently than everything else in C++**. Inadvertent mistakes can be easily made as a result.

Best practice: Prefer **constant variables over object-like macros with substitution text**.

### 7.1.8 *Using constant variables throughout a multi-file program*

In many applications, a given named constant needs to be used throughout your code (not just in one file). 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 them every time** they are needed, 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.

There are **multiple ways to facilitate this** within C++ -- we cover this topic in full detail in lesson: Sharing global constants across multiple files (using inline variables).

### 7.1.9 *Type qualifiers*

A **type qualifier** (sometimes called a **qualifier** for short) is a keyword that is applied to a type that **modifies how that type behaves**. The `const` used to declare a constant variable is called a **const type qualifier** (or **const qualifier** for short).

As of C++23, C++ only has **two type qualifiers**: **const** and **volatile**.

The **volatile** qualifier is used to tell the compiler that an **object may have its value changed at any time**. This rarely-used qualifier disables certain types of optimizations.

In technical documentation, the `const` and `volatile` qualifiers are often referred to as `cv-qualifiers`. The following terms are also used in the C++ standard:

- A **cv-unqualified** type is a type with no type qualifiers (e.g. `int`).
- A **cv-qualified** type is a type with one or more type qualifiers applied (e.g. `const int`).
- A **possibly cv-qualified** type is a type that may be `cv-unqualified` or `cv-qualified`.

These terms are not used much outside of technical documentation, so they are listed here for reference, not as something you need to remember.

## 7.2 Literals

**Literals** are **values** that are **inserted directly into the code**. Literals are sometimes called **literal constants** because their meaning cannot be redefined (5 always means the integral value 5).

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`.

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

### 7.2.1 Literal suffixes

If the default type of a literal is not as desired, **you can change the type of a literal by adding a suffix**. In most cases, suffixes aren't needed (**except for f**). It's important to **know what** the default types (floating-point types are **double** and integral types are **int**) are and what **literal types you are trying to use**. Mistakenly **using the wrong type requires the compiler to do conversion**.

**For example:** `long c { 7 }; // ok: compiler will convert int value 7 to long value 7`

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 <code>std::size_t</code> (C++23)
integral	uz, uZ, Uz, UZ, zu, zU, Zu, ZU	<code>std::size_t</code> (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	<code>std::string</code>
string	sv	<code>std::string_view</code>

The **s** and **sv** suffixes **require an additional line of code to use**. We cover these in further lessons.

**Additional** (rarely used) **suffixes exist** for complex numbers and chrono (time) literals.

Excepting the **f** suffix, **suffixes are most often used in cases where type deduction is involved**.

### 7.2.2 *Suffix casting*

**Most of the suffixes are not case sensitive.**

The exceptions are:

- **s** and **sv** must be lower case.
- **Two consecutive l or L** characters must have the same casing.

Because lower-case L can look like numeric 1 in some fonts, some developers prefer to use upper-case literals. Others use lower case suffixes except for L. **Best practice** here is to **prefer literal suffix L (upper case) over l (lower case)**.

### 7.2.3 *String literals*

In programming, a string is a **collection of sequential characters** used to represent text (such as names, words, and sentences). **The very first C++ program most people write probably printed hello world without using any variables.**

"Hello, world!" is a **string literal**. String literals are placed between double quotes to identify them as strings (as opposed to **char literals**, which are placed **between single quotes**).

Because strings are commonly used in programs, most modern programming languages include a fundamental string data type. **For historical reasons, strings are not a fundamental type in C++**. Rather, they have a strange, complicated type that is hard to work with (we'll cover how/why in a future lesson, once we've covered more fundamentals required to explain how they work). **Such strings are often called C strings or C-style strings, as they are inherited from the C-language.**

There are two non-obvious things worth knowing about C-style string literals:

All **C-style string literals have an implicit null terminator**. Consider a string such as "hello". While this C-style string appears to only have five characters, it **actually has six**: 'h', 'e', 'l', 'l', 'o', and '\0' (a character with **ASCII code 0**). This trailing '\0' character is a special character called a null terminator, and it is used to indicate the end of the string. A string that ends with a null terminator is called a null-terminated string.

The **reason for the null-terminator is also historical**: it can be used to **determine where the string ends**.

Unlike most other literals (**which are values, not objects**), **C-style string literals are const objects** that are created at the start of the program and are **guaranteed to exist for the entirety of the program**. This fact will become important in a few lessons, when we discuss **std::string\_view**.

Unlike C-style string literals, **std::string** and **std::string\_view** literals **create temporary objects**. These temporary objects **must be used immediately**, as they are destroyed at the end of the full expression in which they are created.

#### 7.2.4 *Magic numbers*

A magic number is a **literal** (usually a number) **that either has an unclear meaning or may need to be changed later**.

Best practice: **Avoid magic numbers** in your code (**use constexpr variables instead**).

### 7.3 Numeral systems (decimal, binary, hexadecimal, and octal)

In everyday life, we count using **decimal numbers**, where each numerical digit can be 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Decimal is also called “**base 10**”, because there are 10 possible digits (0 through 9). In this system, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... By default, numbers in C++ programs are assumed to be decimal.

In **binary**, there are only 2 digits: 0 and 1, so it is called “**base 2**”. In binary, we count like this: 0, 1, 10, 11, 100, 101, 110, 111, ...

Decimal and binary are two examples of **numeral systems**, which is a **fancy name for a collection of symbols** (e.g. digits) used to represent numbers. There are **4 main numeral systems** available in C++. In order of popularity, these are: decimal (**base 10**), binary (**base 2**), hexadecimal (**base 16**), and octal (**base 8**).

Numeral systems **prefixes**:

**0**      **Octal**

**0x**     **Hexadecimal**

**0b**     **Binary**

```
int main()
{
    int hex = 0x0008;
    std::cout << hex << "\n";

    int oct = 010;
    std::cout << oct << "\n";

    int bin = 0b1000;
    std::cout << bin << "\n";

    return 0;
}
```

All of the following cout statements print the number 8.

### 7.3.1 Digit separators

Because long literals can be hard to read, C++14 also adds the ability to use a **quotation mark (')** as a **digit separator**.

```
int bin { 0b1011'0010 }; // assign binary 1011 0010 to the variable
long value { 2'132'673'462 }; // much easier to read than 2132673462
```

### 7.3.2 Outputting values in decimal, octal, or hexadecimal (and binary)

By default, C++ outputs values in decimal. However, you can change the output format via use of the **std::dec**, **std::oct**, and **std::hex** I/O manipulators. Note that once applied, the I/O manipulator remains set for future output until it is changed again.

#### 7.3.2.1 Outputting values in binary

**Outputting values in binary is a little harder**, as **std::cout** doesn't come with this capability built-in. Fortunately, the **C++ standard library includes a type called **std::bitset**** that will do this for us (in the **<bitset>** header).

To use **std::bitset**, we can **define a **std::bitset** variable** and tell **std::bitset** how many bits we want to store. The **number of bits must be a compile-time constant**. **std::bitset** can be initialized with an integral value (in any format, including decimal, octal, hex, or binary).

```
1 #include <bitset> // for std::bitset
2 #include <iostream>
3
4 int main()
5 {
6     // std::bitset<8> means we want to store 8 bits
7     std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101
8     std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101
9
10    std::cout << bin1 << '\n' << bin2 << '\n';
11    std::cout << std::bitset<4>{ 0b1010 } << '\n'; // create a temporary std::bitset and print it
12
13    return 0;
14 }
```

Like the code above has shown. **Std::bitset<n>** creates a **temporary (unnamed) std::bitset object with n amount of bits**, initializes it with the inserted binary literal, prints the value in binary, and then discards the temporary object.



## 7.4 Constant expressions and compile-time optimization (run or compile-time)

### The as-if rule:

In C++, compilers are given a lot of leeway to optimize programs. The as-if rule says that 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**".

For advanced readers: There is **one exception to the as-if rule: unnecessary calls to a copy constructor can be elided (omitted) even if those copy constructors have observable behavior**. We cover this topic in lesson: Class initialization and **copy elision**.

Exactly how a compiler optimizes a given program is up to the compiler itself. **However, there are things we can do to help the compiler optimize better.**

### 7.4.1 An optimization opportunity

Consider the following program with output 7:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x { 3 + 4 };
6      std::cout << x << '\n';
7
8      return 0;
9  }
```

There's an interesting optimization possibility hidden within this program.

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.**

**Note:** Some compilers use **constant folding**, an optimization technique where the **compiler evaluates constant expressions at compile-time** and replaces them with their results. This reduces the need for calculations during run-time, thereby improving performance.

### 7.4.2 *Compile-time evaluation of expressions*

**Modern C++ compilers are able to evaluate some expressions at compile-time** (constant folding). When this occurs, the compiler can replace the expression with the result of the expression. The resulting executable **no longer needs to spend CPU cycles calculating** at runtime! Even better, we don't need to do anything to enable this behavior (**besides have optimizations turned on**).

### 7.4.3 *Constant expressions*

One kind of **expression that can always be evaluated at compile time** is called a “**constant expression**”. The precise definition of a constant expression is complicated, so we'll take a simplified view:

A constant expression is an **expression that contains only compile-time constants and operators/functions that support compile-time evaluation**.

A **compile-time constant** is a constant whose value must be known at compile time. This includes:

- **Literals**
- **Const integral variables with a constant expression initializer** (historical exception)
- **Constexpr variables and functions** (preferred in C++)
- **Preprocessor macros**
- **Non-type template parameters**
- **Enumerators**

Const variables that are not compile-time constants are sometimes called **runtime constants**. Runtime constants **cannot be used in a constant expression**.

**Const non-integral variables are always runtime constants** (even if they have a constant expression initializer). If you need such variables to be compile-time constants, **define them as constexpr variables instead**.

The most common type of **operators and functions that support compile-time evaluation** include:

- **Arithmetic operators** with operands that are compile-time constants (e.g.  $1 + 2$ )
- **Constexpr and consteval functions** (we'll discuss these later in the chapter)

In the following example, we **identify the constant expressions and non-constant expressions**. We also identify which variables are non-constant, runtime constant, or compile-time constant.

```

1  #include <iostream>
2
3  int getNumber()
4  {
5      std::cout << "Enter a number: ";
6      int y{};
7      std::cin >> y;
8
9      return y;
10 }
11
12 int main()
13 {
14     // Non-const variables:
15     int a { 5 };           // 5 is a constant expression
16     double b { 1.2 + 3.4 }; // 1.2 + 3.4 is a constant expression
17
18     // Const integral variables with a constant expression initializer
19     // are compile-time constants:
20     const int c { 5 };     // 5 is a constant expression
21     const int d { c };     // c is a constant expression
22     const long e { c + 2 }; // c + 2 is a constant expression
23
24     // Other const variables are runtime constants:
25     const int f { a };     // a is not a constant expression
26     const int g { a + 1 }; // a + 1 is not a constant expression
27     const long h { a + c }; // a + c is not a constant expression
28     const int i { getNumber() }; // getNumber() is not a constant expression
29
30     const double j { b };  // b is not a constant expression
31     const double k { 1.2 }; // 1.2 is a constant expression
32
33     return 0;
34 }

```

An expression that is **not a constant expression** is sometimes called a **runtime expression**. For example, `std::cout << x << '\n'` is a runtime expression, both because `x` is not a compile-time constant, and because `operator<<` doesn't support compile-time evaluation when used for output (since **output can't be done at compile-time**).

#### 7.4.4 *Why we care about constant expressions*

**Constant expressions** are **useful for** (at least) **three reasons**:

- **Constant expressions are always eligible for compile-time evaluation**, meaning they are more likely to be **optimized at compile-time**. This produces **faster and smaller code**.
- With runtime expressions, only the type of the expression is known at compile-time. **With constant expression, both the type AND the value of the expression is known at compile-time**. This **allows us to do compile-time sanity checking** of those values. If such a value does not meet our requirements, we can fail the build, **allowing us to identify and fix the issue immediately**. The result is **code that is safer, easier to test, and more difficult to misuse**.
- **Certain C++ features** that we'll cover in future lessons **require constant expressions** (see below)

**For advanced readers:** There are a few common cases where a constant expression is required:

- The initializer of a constexpr variable.
- A non-type template argument.
- The defined length of a std::array

**Key insight:**

**Constant expressions** (including constexpr variables and constexpr functions, which we'll cover shortly) can **help produce code that is faster, smaller, and safer**. **Some C++ features require constant expressions**.

### 7.4.5 When are constant expressions evaluated?

The compiler is only required to evaluate constant expressions at compile-time in contexts that **require a constant expression (such as the initializer of a compile-time constant)**. In contexts that do not require a constant expression, the **compiler may choose** whether to evaluate a constant expression at compile-time or at runtime.

In short: **Using the const modifier under the conditions that the compiler can evaluate the expression at compile-time enforces compile-time evaluation.**

Even when not required to do so, **modern compilers will usually evaluate a constant expression at compile-time** because it is an easy optimization and more performant to do so.

#### 7.4.5.1 Partial optimization of constant subexpressions (constant folding)

Now consider the following example:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << 3 + 4 << '\n';
6
7     return 0;
8 }
```

The **full expression** `std::cout << 3 + 4 << '\n';` is a **runtime expression** because **output can only be done at runtime**. But notice that the **full expression contains constant subexpression 3 + 4**.

**Compilers** have long been **able to optimize constant subexpressions**, even when the full expression is a runtime expression. This optimization process is called “**constant folding**”, and is **allowed under the as-if rule**.

### 7.4.5.2 Optimization of non-constant expressions

Compilers are even capable of optimizing non-constant expressions or subexpressions in certain cases. Let's revisit a prior example:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 7 };           // x is non-const
6     std::cout << x << '\n'; // x is a non-constant subexpression
7
8     return 0;
9 }
```

When `x` is initialized, the value 7 will be stored in the memory allocated for `x`. Then on the next line, the program will go out to memory again to fetch the value 7 so it can be printed. **Even though `x` is non-const, a smart compiler might realize that `x` will always evaluate to 7 in this particular program, and under the as-if rule, optimize the program to this:**

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 7 };
6     std::cout << 7 << '\n';
7
8     return 0;
9 }
```

Since `x` is no longer used in the program, the compiler could go one step further and optimize the program to this:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << 7 << '\n';
6
7     return 0;
8 }
```

In this version, the **variable `x` was removed completely** (because it was not used, and thus not needed). When a variable is removed from a program, **we say the variable has been optimized out (or optimized away)**.

However, since `x` is non-const, such optimizations require the compiler to realize that the value of `x` actually doesn't change (even though it could). **Whether the compiler realizes this comes down to how complex the program is and how sophisticated the compiler's optimization routines are.**

### 7.4.5.3 Const variables are easier to optimize

Now let's consider this similar example:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     const int x { 7 }; // x is now const
6 |     std::cout << x << '\n';
7 |
8 |     return 0;
9 | }
```

In this version, the **compiler no longer has to infer that x won't change**. Because **x is now const**, the compiler now has a guarantee that x can't be changed after initialization. This makes it easier for the compiler to understand that it can safely optimize x out of this program, and therefore it is more likely to do so.

Ranking variables by the likelihood of the compiler being able to optimize them:

- Compile-time constant variables (always eligible to be optimized)
- Runtime constant variables
- Non-const variables (likely optimized in simple cases only)

#### 7.4.6 *Optimization can make programs harder to debug*

**When the compiler optimizes a program, variables, expressions, statements, and function calls may be rearranged, altered, replaced with a value, or even removed entirely.** Such changes can make it hard to debug a program effectively.

**At runtime, it can be hard to debug compiled code that no longer correlates very well with the original source code.** For example, if you try to watch a variable that has been optimized out, the debugger won't be able to locate the variable. If you try to step into a function that has been optimized away, the debugger will simply skip over it. **So, if you are debugging your code and the debugger is behaving strangely, this is the most likely reason.**

**At compile-time, we have little visibility and few tools to help us understand what the compiler is even doing.** If a variable or expression is replaced with a value, and that value is wrong, how do we even go about debugging it? This is an ongoing challenge.

To help minimize such issues, **debug builds will typically turn down (or turn off) optimizations, so that the compiled code will more closely match the source code.**



## 7.5 Constexpr variables (compile-time)

### 7.5.1 *The compile-time const challenge*

We discussed how **one way to make a compile-time constant variable is to use the `const` keyword**. If the `const` variable has an integral type and a constant expression initializer, it is a compile-time constant. **All other `const` variables are treated as runtime constants**.

However, **this method has two challenges**:

First, when using `const`, **our integral variables could end up as either a compile-time const or a runtime const**, depending on whether the initializer is a constant expression or not. **In some cases, this can make it hard to tell whether the `const` variable is actually a compile-time constant or not**. Consider a scenario where you're reading code and **see a `const` variable initialized through another variable that you don't have access to**. This might or might not be `const` depending on the referenced variable.

Second, this use of `const` to create compile-time constant variables **does not extend to non-integral variables**. And there are many cases where we would like non-integral variables to be compile-time constants too.

### 7.5.2 *The `constexpr` keyword*

Fortunately, 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**.

### 7.5.3 The meaning of `const` vs `constexpr` for variables

For variables:

- **`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`.

Best practice:

- Any constant **variable whose initializer is a constant expression should be declared as `constexpr`**.
- Any constant **variable whose initializer is not a constant expression** (making it a runtime constant) **should be declared as `const`**.

**Caveat:** In the future we will discuss some **types that are not fully compatible with `constexpr`** (including `std::string`, `std::vector`, and other types that use dynamic memory allocation). For constant objects of these types, either **use `const` instead of `constexpr`**, or pick a different type that is `constexpr` compatible (e.g. `std::string_view` or `std::array`).

### 7.5.4 `Const` and `constexpr` function parameters

**Normal function calls are evaluated at runtime**, with the supplied arguments being used to **initialize the function's parameters**. Because the initialization of function parameters happens at runtime, this leads to **two consequences**:

- **`const` function parameters are treated as runtime constants** (even when the supplied argument is a compile-time constant).
- Function parameters **cannot be declared as `constexpr`**, since their **initialization value isn't determined until runtime**.

### 7.5.5 *Nomenclature recap*

Compile-time constant

- A value or **non-modifiable object whose value must be known at compile time** (e.g. literals and constexpr variables).

Constexpr

- **Keyword** that **declares variables as compile-time constants** (and functions that can be evaluated at compile-time). Informally, shorthand for “constant expression”.

Constant expression

- An **expression that contains only compile-time constants** and operators/functions that support compile-time evaluation.

Runtime expression

- An expression that is **not a constant expression**.

Runtime constant

- A value or **non-modifiable object that is not a compile-time constant**.

## 7.6 The conditional operator

Operator	Symbol	Form	Meaning
Conditional	?:	<code>c ? x : y</code>	If conditional <code>c</code> is true then evaluate <code>x</code> , otherwise evaluate <code>y</code>

The **conditional operator** (?:) (also sometimes called the **arithmetic if** operator) is a **ternary operator** (an **operator that takes 3 operands**). Because it has historically been C++'s only ternary operator, it's also sometimes referred to as "**the ternary operator**".

The `?:` operator provides a **shorthand method** for doing a particular type of **if-else statement**.

Syntax: `condition ? expression_if_true : expression_if_false`

### 7.6.1 The conditional operator evaluates as an expression

Because the operands of the conditional operator are **expressions rather than statements**, the **conditional operator can be used in places where an expression is required**.

For example, when initializing a variable:

```
constexpr bool inBigClassroom { false };
constexpr int classSize { inBigClassroom ? 30 : 20 };
std::cout << "The class size is: " << classSize << '\n';
```

Best practice:

- Parenthesize the entire conditional operator when used in a compound expression.
- For readability, consider parenthesizing the condition if it contains any operators (other than the function call operator).

### 7.6.2 *Parenthesizing the conditional operator*

Because C++ prioritizes the evaluation of most operators above the evaluation of the conditional operator, it's quite easy to write expressions using the conditional operator that don't evaluate as expected.

For this reason, the conditional operator should be parenthesized as follows:

- **Parenthesize the entire conditional operator when used in a compound expression** (an expression with other operators).
- For readability, **consider parenthesizing the condition** if it contains any operators (other than the function call operator).
- The **operands** of the conditional operator **do not need to be parenthesized**.

### 7.6.3 *The type of the expressions must match or be convertible*

To comply with C++'s type checking rules, **one of the following must be true**:

- The **type of the second and third operand must match**.
- The **compiler must be able to find a way to convert one or both of the second and third operands to matching types**. The conversion rules the compiler uses are fairly complex and may yield surprising results in some cases.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << (true ? 1 : 2) << '\n';    // okay: both operands have matching type int
6
7      std::cout << (false ? 1 : 2.2) << '\n'; // okay: int value 1 converted to double
8
9      std::cout << (true ? -1 : 2u) << '\n';  // surprising result: -1 converted to unsigned int, result out of range
10
11     return 0;
12 }
```

In general, it's **okay to mix operands with fundamental types** (excluding mixing signed and unsigned values). If either operand is not a fundamental type, it's **generally best to explicitly convert one or both operands to a matching type yourself** so you know exactly what you'll get.

The conditional operator is **most useful when** doing one of the following:

- **Initializing** an object **with one of two values**.
- **Assigning one of two values** to an object.
- **Passing one of two values** to a function.
- **Returning one of two values** from a function.
- **Printing one of two values**.

Best practice:

Prefer to **avoid the conditional operator in complicated expressions**.

## 7.7 Inline functions and variables (may or may not inline)

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 function** (and possibly sub-functions) to handle the task.

Writing **functions provides many potential benefits**, as code in a function:

- Is **easier to read** and understand in the context of the overall program.
- Is **easier to use**, as you can call the function without understanding how it is implemented.
- Is **easier to update**, as the code in a function can be updated in one place.
- Is **easier to reuse**, as functions are naturally modular.

However, **one downside** of using a function is that **every time a function is called, there is a certain amount of performance overhead (“overhead cost”)** that occurs.

### 7.7.1 *Function overhead*

When a call to a function is encountered the CPU has to:

1. Store the address of the current instruction (bookmark **to return to later**)
2. Store values of various CPU registers (to **restore later**)
3. Function **parameters get instantiated and then initialized**
4. Execution path makes a **jump into the function**
5. Jump back to location of the call instruction
6. Return value has to be copied for use

For **functions that are large and/or perform complex tasks**, the **overhead** of the function call is **typically insignificant** compared to the amount of time the function takes to run.

However, for small functions, the overhead costs can be larger than the time needed to actually execute the function's code! In cases where a small function is called often, **using a function can result in a significant performance penalty over writing the same code in-place.**

### 7.7.2 *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.**

**Function calls will be replaced by the code in the body of the called function** (with the **value of the arguments substituted for the parameters**). This **allows us to avoid the overhead** of those calls, **while preserving the results** of the code.

### 7.7.3 *The performance of inline code*

Beyond **removing the cost of function call**, inline expansion can also allow the compiler to optimize the resulting code more efficiently. For example: the code that gets inlined could **potentially be optimized into a constant expression**.

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).

### 7.7.4 *When inline expansion occurs*

Every function falls into one of two categories, where **calls to the function**:

- **May be expanded** (most functions are in this category).
- **Can't be expanded**.

**Most functions** fall into the “**may**” category: their function calls **can be expanded if and when it is beneficial to do so**. For functions in this category, a modern **compiler will assess each function and each function call** to make a determination about whether that particular function call would benefit from inline expansion. A **compiler might decide to expand none, some, or all of the function calls** to a given function.

In short: **Modern optimizing compilers make the decision about when functions should be expanded inline**.



### 7.7.5 *The inline keyword, historically*

Historically, **compilers either didn't have the capability to determine whether inline expansion would be beneficial, or were not very good at it.** For this reason, **C++ provided the keyword inline**, which was **originally intended to be used as a hint to the compiler** that a function would (probably) benefit from being expanded inline.

**A function that is declared using the inline keyword is called an inline function.**

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 -- the compiler is completely free to ignore a request to inline a function.** This is likely to be the result if you try to inline a lengthy function! 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.

Best practice:

**Do not use the inline** keyword to request inline expansion for your functions. Modern C++ compilers will do the work for you either way.

### 7.7.6 *The inline keyword, modernly*

In previous chapters, we mentioned that you should not implement functions (with external linkage) in header files, because when those headers are included into multiple .cpp files, the function definition will be copied into multiple .cpp files. These files will then be compiled, and the linker will throw an error because it will note that you've defined the same function more than once, which is a violation of the one-definition rule.

**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). The definition can occur after the point of use if a forward declaration is also provided. **Only one such definition can occur per translation unit, otherwise a compilation error will occur.**
- Every **definition for an inline function must be identical, otherwise undefined behavior will result.**

**Important rule:** The **compiler needs to** be able to **see the full definition** of an inline function **wherever it is used**, and **all such definitions must be identical** (or undefined behavior will result).

The **linker will consolidate all inline function definitions for an identifier into a single definition** (thus still meeting the requirements of the one definition rule).

main.cpp:

```
1 | #include <iostream>
2 |
3 | double circumference(double radius); // forward declaration
4 |
5 | inline double pi() { return 3.14159; }
6 |
7 | int main()
8 | {
9 |     std::cout << pi() << '\n';
10 |    std::cout << circumference(2.0) << '\n';
11 |
12 |    return 0;
13 | }
```

math.cpp

```
1 | inline double pi() { return 3.14159; }
2 |
3 | double circumference(double radius)
4 | {
5 |     return 2.0 * pi() * radius;
6 | }
```

Notice that both files have a definition for function `pi()` -- however, **because this function has been marked as inline, this is acceptable**, and the linker will **de-duplicate** them. If you remove the `inline` keyword from both definitions of `pi()`, you'll get an ODR violation (as duplicate definitions for non-inline functions are disallowed).

Inline functions are **typically defined in header files**, where they can be **#included into the top of any code file that needs to see the full definition of the identifier**. This ensures that all inline definitions for an identifier are identical.

pi.h:

```
1 | #ifndef PI_H
2 | #define PI_H
3 |
4 | inline double pi() { return 3.14159; }
5 |
6 | #endif
```

```
main.cpp:
1  #include "pi.h" // will include a copy of pi() here
2  #include <iostream>
3
4  double circumference(double radius); // forward declaration
5
6  int main()
7  {
8      std::cout << pi() << '\n';
9      std::cout << circumference(2.0) << '\n';
10
11     return 0;
12 }
```

Math.cpp

```
1  #include "pi.h" // will include a copy of pi() here
2
3  double circumference(double radius)
4  {
5      return 2.0 * pi() * radius;
6  }
```

This is particularly useful for **header-only libraries**, which are one or more header files that implement some capability (no .cpp files are included). Header-only libraries are popular because there are no source files that need to be added to a project to use them and nothing that needs to be linked. You simply #include the header-only library and then can use it.

**Best practice:**

**Avoid the use of the inline keyword unless you have a specific, compelling reason to do so** (e.g. you're defining those functions or variables in a header file).

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

There are a few good reasons:

- **It can increase your compile times.** When a function in a code file changes, only that code 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 be recompiled.** On large projects, this can have a drastic impact.

- It can lead to more naming collisions since you'll end up with more code in a single code file.

### 7.7.8 *Inline variables*

In the above example, `pi()` was written as a function that returns a constant value. **It would be more straightforward if `pi` were implemented as a `(const)` variable instead.** However, prior to C++17, there were some obstacles and inefficiencies in doing so.

**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).

### 7.7.9 *Summary and implicit inlines*

Functions calls contain **overhead** and this increases execution time. Only consider inline if you have a small function (**small enough to outweigh the overhead cost**) that you know will be called many times.

The inline keyword is **no more than the suggestion** (for the compiler) to inline something. The **compiler can still inline or not inline the function**.

Also note that **inline expansions have their own potential cost**:

- If not done correctly it **could result in either performance improvements, reductions or no changes to the performance at all**.
- It can **increase compile times**
- Once an inline function **has changes in a header file, every translation unit** using that file **has to be recompiled**
- In the long run **you could end up with naming collisions** because the **amount of code across files increases**

Historically compilers didn't know how to pick, choose or detect functions the could be expanded with inline. Hence why **historically inline was used to hint the compiler to inline a function**.

However, **as of today** inline is **used to allow multiple definitions**. So, inline functions are actually **allowed to be defined in multiple translation units without violating ODR** (the linker will actually de-duplicate them).

There's still **rules** to how many times it can be included:

- Once per translation unit
- Every definition of the inline function has to be identical

In general, it is **best practice to avoid the inline keyword** unless you have a specific reason to do so.

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).

#### **Advanced:**

The following are **implicitly inlined functions**:

- Functions defined inside a class, struct, or union type definition
- **constexpr / consteval functions**
- Functions implicitly instantiated from function templates

The following are **implicitly inlined variables**:

- Static **constexpr data members**

**Unlike constexpr functions, constexpr variables are not inline by default** (excepting those noted above)!

## 7.8 constexpr functions (run or compile-time) and consteval functions (compile-time)

Earlier we introduced the **constexpr** keyword, which we used to create **compile-time (symbolic) constants**. We also introduced **constant expressions**, which are **expressions that can be evaluated at compile-time rather than runtime**.

**One challenge with constant expressions is that a function call to a normal function is allowed in constant expressions.** This means we cannot use such function calls anywhere a constant expression is required.

Consider the following program:

```
1  #include <iostream>
2
3  int main()
4  {
5      constexpr double radius { 3.0 };
6      constexpr double pi { 3.14159265359 };
7      constexpr double circumference { 2.0 * radius * pi };
8
9      std::cout << "Our circle has circumference " << circumference << "\n";
10
11     return 0;
12 }
```

Having a complex initializer for circumference isn't great (and requires us to instantiate two supporting variables, radius and pi). Making it into a function instead results in much cleaner code.

```
constexpr double circumference { calcCircumference(3.0) }; // compile error
```

Unfortunately, it also **doesn't compile**. Cpp variable circumference **requires that its initializer is a constant expression**, and the call **calcCircumference()** **isn't a constant expression**. In this particular case, **we could make circumference non-constexpr**, and the program would compile. While we'd lose the benefits of constant expressions, at least the program would run.

However, there are other cases in C++ (which we'll introduce in the future) where we do not have alternate options available, and only a constant expression will do. **In those cases, we'd really like to be able to use functions**, but calls to normal functions just won't work.

**So, what are we to do? → Cpp or consteval functions**

### 7.8.1 *Constexpr functions can be used in constant expressions*

Here's the same example as above, but using a constexpr function:

```
1  #include <iostream>
2
3  constexpr double calcCircumference(double radius) // now a constexpr function
4  {
5      constexpr double pi { 3.14159265359 };
6      return 2.0 * pi * radius;
7  }
8
9  int main()
10 {
11     constexpr double circumference { calcCircumference(3.0) }; // now compiles
12
13     std::cout << "Our circle has circumference " << circumference << "\n";
14
15     return 0;
16 }
```

Because calcCircumference() is now a constexpr function, it can be used in a constant expression, such as the initializer of circumference.

### 7.8.2 *Constexpr functions can be evaluated at compile time*

In Constant expressions and compile-time optimization, we noted that in contexts that require a constant expression (such as the initialization of a constexpr variable), a constant expression is required to evaluate at compile-time. **If a required constant expression contains a constexpr function call, that constexpr function call must evaluate at compile-time.**

In our example above, variable circumference is constexpr and thus requires a constant expression initializer. Since calcCircumference() is part of this required constant expression, calcCircumference() must be evaluated at compile-time.

**When a function call is evaluated at compile-time, the compiler will calculate the return value of the function call at compile-time, and then replace the function call with the return value.**



To evaluate at compile-time, two other things must also be true:

- The call to the constexpr function must have **arguments that are known at compile time** (e.g. are **constant expressions**).
- **All statements and expressions within the constexpr function must be evaluable at compile-time.**

When a constexpr (or consteval) function is being evaluated at compile-time, **any other functions it calls are required to be evaluated at compile-time (otherwise the initial function would not be able to return a result at compile-time).**

**For advanced readers:** There are some other lesser encountered criteria as well. These can be found on the cppreference website.

### 7.8.3 *Constexpr functions can also be evaluated at runtime*

Constexpr functions can also be evaluated at runtime, in which case they will return a non-constexpr result.

**Key insight:** When a constexpr function evaluates at runtime, it evaluates just like a normal (non-constexpr) function would. In other words, the **constexpr has no effect in this case.**

**Allowing functions with a constexpr return type to be evaluated at either compile-time or runtime was allowed so that a single function can serve both cases.**

Otherwise, you'd need to have separate functions (a function with a constexpr return type, and a function with a non-constexpr return type). **This would not only require duplicate code, the two functions would also need to have different names!**

Note that even **non-constexpr functions could be evaluated at compile-time** under the as-if rule!

**Key insight:**

Put another way, we can categorize the **likelihood that a function will actually be evaluated at compile-time** as follows:

**Always** (required by the standard):

- Constexpr function is called **where constant expression is required**.
- Constexpr function is called **from other function being evaluated at compile-time**.

**Probably** (there's little reason not to):

- Constexpr function is called **where constant expression isn't required, all arguments are constant expressions**.

**Possibly** (if optimized under the as-if rule):

- Constexpr function is called **where constant expression isn't required, some arguments are not constant expressions but their values are known at compile-time**.
- **Non-constexpr function** capable of being evaluated at compile-time, **all arguments are constant expressions**.

**Never** (not possible):

- Constexpr function is called **where constant expression isn't required, some arguments have values that are not known at compile-time**.

Note that **your compiler's optimization level setting may have an impact on whether it decides to evaluate a function at compile-time or runtime**. This also means that your compiler **may make different choices for debug vs. release builds** (as debug builds typically have optimizations turned off).

For example, both gcc and Clang will not compile-time evaluate a constexpr function called where a constant expression isn't required unless the compiler told to optimize the code (e.g. using the -O2 compiler option).

For advanced readers: The **compiler might also choose to inline a function call, or even optimize a function call away entirely**. Both of these can affect when (or if) the content of the function call are evaluated.

#### 7.8.4 *What about `std::is_constant_evaluated` or `if constexpr`? (Advanced)*

C++ does not currently provide any reliable mechanisms to determine if a constexpr function call is evaluating at compile-time or runtime.

Neither of these capabilities tell you whether a function call is evaluating at compile-time or runtime.

`std::is_constant_evaluated()` (defined in the `<type_traits>` header) **returns a bool indicating whether the current function is executing in a constant-evaluated context**. A constant-evaluated context (also called a **constant context**) is **defined as one in which a constant expression is required** (such as the initialization of a constexpr variable). So, in cases where the compiler is required to evaluate a constant expression at compile-time `std::is_constant_evaluated()` will true as expected.

**However, the compiler may also choose to evaluate a constexpr function at compile-time in a context that does not require a constant expression.** In such cases, `std::is_constant_evaluated()` will return false even though the function did evaluate at compile-time. So `std::is_constant_evaluated()` really means “the compiler is being forced to evaluate this at compile-time”, not “this is evaluating at compile-time”.

While this may seem strange, there are several reasons for this:

- As the paper that proposed this feature indicates, the standard doesn’t actually make a distinction between “compile time” and “runtime”. Defining behavior involving that distinction would have been a larger change.
- If `std::is_constant_evaluated()` returned true when a function was evaluated at compile-time for any reason, the optimizer (acting under the as-if rule) would potentially be able to change the behavior of a program by optionally evaluating a function at compile time (which would be a violation of the as-if rule). While this could be addressed in various ways, those involve adding additional complexity to the optimizer and/or limiting its ability to optimize certain cases.

Introduced in C++23, `if constexpr` is a replacement for `if (std::is_constant_evaluated())` that provides a nicer syntax and fixes some other issues. However, it evaluates the same way.

### 7.8.5 *Forcing a constexpr function to be evaluated at compile-time*

There is no way to tell the compiler that a constexpr function should prefer to evaluate at **compile-time whenever it can** (e.g. in cases where the return value of a constexpr function is used in a non-constant expression).

However, we can force a constexpr function that is eligible to be evaluated at compile-time to **actually evaluate at compile-time by ensuring the return value is used where a constant expression is required**. This needs to be done on a per-call basis.

The most common way to do this is to **use the return value to initialize a constexpr variable** (this is why we've been using variable 'g' in prior examples). **Unfortunately, this requires introducing a new variable into our program just to ensure compile-time evaluation, which is ugly and reduces code readability.**

### 7.8.6 *Consteval functions (compile-time)*

C++20 introduces the keyword **constexpr**, which is used to indicate that a function must evaluate at compile-time, otherwise a compile error will result. Such functions are called **immediate functions**.

Best practice: Use **constexpr** if you have a function that must evaluate at compile-time for some reason (e.g. because it does something that can only be done at compile time).

Perhaps **surprisingly, the parameters of a constexpr function are not constexpr** (even though constexpr functions can only be evaluated at compile-time). This decision was made for the sake of consistency.

### 7.8.7 Using consteval to make constexpr execute at compile-time

The **downside** of consteval functions is that **such functions can't evaluate at runtime**, making them **less flexible than constexpr functions**, which can do either. Therefore, it would still be useful to have a convenient way to force constexpr functions to evaluate at compile-time (even when the return value is being used where a constant expression is not required), so that we could have compile-time evaluation when possible, and runtime evaluation when we can't.

Consteval functions provides a way to make this happen, using a neat helper function:

```

1  #include <iostream>
2
3  // Uses abbreviated function template (C++20) and 'auto' return type to make this function work with any type of value
4  // See 'related content' box below for more info (you don't need to know how these work to use this function)
5  consteval auto compileTimeEval(auto value)
6  {
7      return value;
8  }
9
10 constexpr int greater(int x, int y) // function is constexpr
11 {
12     return (x > y ? x : y);
13 }
14
15 int main()
16 {
17     std::cout << greater(5, 6) << '\n'; // may or may not execute at compile-time
18     std::cout << compileTimeEval(greater(5, 6)) << '\n'; // will execute at compile-time
19
20     int x { 5 };
21     std::cout << greater(x, 6) << '\n'; // we can still call the constexpr version at runtime if we wish
22
23     return 0;
24 }
```

This works because consteval functions require constant expressions as arguments -- therefore, if we use the return value of a constexpr function as an argument to a consteval function, the constexpr function must be evaluated at compile-time! The consteval function just returns this argument as its own return value, so the caller can still use it.

Note that the **consteval function returns by value**. While this might be inefficient to do at runtime (if the value was some type that is expensive to copy, e.g. std::string), in a compile-time context, it doesn't matter because the entire call to the consteval function will simply be replaced with the calculated return value.

### 7.8.8 *Constexpr/constexpr functions can use non-const local variables*

Within a constexpr or constexpr function, we can use local variables that are not constexpr, and the value of these variables can be changed.

```

2
3  constexpr int doSomething(int x, int y) // function is constexpr
4  {
5      x = x + 2;      // we can modify the value of non-const function parameters
6
7      int z { x + y }; // we can instantiate non-const local variables
8      if (x > y)
9          z = z - 1;  // and then modify their values
10
11      return z;
12  }
13

```

When such functions are evaluated at compile-time, the compiler will essentially “execute” the function and return the calculated value.

### 7.8.9 *Constexpr/constexpr functions can use function parameters and local variables as arguments in constexpr function calls*

Above, we noted, “When a constexpr (or constexpr) function is being evaluated at compile-time, any other functions it calls are required to be evaluated at compile-time.”

Perhaps surprisingly, a constexpr or constexpr function can use its function parameters (which aren’t constexpr) or even local variables (which may not be const at all) as arguments in a constexpr function call. When a constexpr or constexpr function is being evaluated at compile-time, the value of all function parameters and local variables **must be known to the compiler** (otherwise it couldn’t evaluate them at compile-time). Therefore, in this specific context, C++ allows these values to be used as arguments in a call to a constexpr function, and that constexpr function call can still be evaluated at compile-time.

```

1  #include <iostream>
2
3  constexpr int goo(int c) // goo() is now constexpr
4  {
5      return c;
6  }
7
8  constexpr int foo(int b) // b is not a constant expression within foo()
9  {
10     return goo(b);      // if foo() is resolved at compile-time, then 'goo(b)' can also be resolved at compile-time
11 }
12
13 int main()
14 {
15     std::cout << foo(5);
16
17     return 0;
18 }

```

In the above example, `foo(5)` may or may not be evaluated at compile time. If it is, then the compiler knows that `b` is 5. And even though `b` is not `constexpr`, the compiler can treat the call to `goo(b)` as if it were `goo(5)` and evaluate that function call at compile-time. If `foo(5)` is instead resolved at runtime, then `goo(b)` will also be resolved at runtime.

#### 7.8.10 Can a `constexpr` function call a non-`constexpr` function?

The answer is yes, but **only when the `constexpr` function is being evaluated in a non-constant context**. A non-`constexpr` function may not be called when a `constexpr` function is evaluating in a constant context (because then the `constexpr` function wouldn't be able to produce a compile-time constant value), and doing so will produce a compilation error.

Calling a non-`constexpr` function is allowed so that a `constexpr` function can do something like this:

```
1 #include <type_traits> // for std::is_constant_evaluated
2
3 constexpr int someFunction()
4 {
5     if (std::is_constant_evaluated()) // if evaluating in constant context
6         return someConstexprFcn();
7     else
8         return someNonConstexprFcn();
9 }
```

Now consider this variant:

```
1 constexpr int someFunction(bool b)
2 {
3     if (b)
4         return someConstexprFcn();
5     else
6         return someNonConstexprFcn();
7 }
```

This is **legal as long as `someFunction(false)` is never called in a constant expression**.

For best results, we'd advise the following:

- **Avoid calling non-`constexpr` functions from within a `constexpr` function** if possible.
- If your `constexpr` function requires **different behavior for constant and non-constant contexts**, **conditionalize the behavior with `if (std::is_constant_evaluated())`** (in C++20) or **`if constexpr`** (C++23 onward).
- **Always test your `constexpr` functions in a constant context**, as they may work when called in a non-constant context but fail in a constant context.

### 7.8.11 *Why not constexpr every function?*

There are a few reasons you may not want to constexpr a function:

- constexpr signals that a function can be used in a constant expression. **If your function cannot be evaluated as part of a constant expression, it should not be marked as constexpr.**
- **constexpr is part of the interface of a function.** Once a function is made constexpr, it can be called by other constexpr functions or used in contexts that require constant expressions. **Removing the constexpr later will break such code.**
- **constexpr makes functions harder to debug** because you **can't inspect them at runtime.**

### 7.8.12 *Why constexpr a function when it is not actually evaluated at compile-time?*

New programmers sometimes ask, “why should I constexpr a function when it is only evaluated at runtime in my program (e.g. **because the arguments in the function call are non-const**)”?

There are a few reasons:

- There's **little downside to using constexpr**, and it **may help the compiler optimize your program** to be smaller and faster.
- Just because you're not calling the function in a compile-time evaluable context right now **doesn't mean you won't call it in such a context when you modify or extend your program.** And if you haven't constexpr'd the function already, you may not think to when you do start to call it in such a context, and then you'll miss out on the performance benefits. Or you may be forced to constexpr it later when you need to use the return value in a context that requires a constant expression somewhere.
- **Repetition helps ingrain best practices.**

#### **Best practice:**

Unless you have a specific reason not to, a **function that can evaluate as part of a constant expression should be made constexpr.**

**A function that cannot be evaluated as part of a constant expression should not be marked as constexpr.**



## 7.9 Introduction to `std::string`

“This is a C-style string literal!”

While **C-style string literals** are fine to use, **C-style string variables behave oddly**, are **hard to work with** (e.g. you **can’t use assignment to assign a C-style string variable a new value**), and are **dangerous** (e.g. **if you copy a larger C-style string into the space allocated for a shorter C-style string, undefined behavior will result**). 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.

### 7.9.1 *Introducing `std::string`*

The easiest way to work with strings and string objects in C++ is via the **`std::string` type**, which lives in the **`<string>` header**.

We can **create objects of type `std::string` just like other objects**.

Assignment, output behave just like string types would in most other languages.

**Key insight:**

**If `std::string` doesn’t have enough memory to store a string, it will request additional memory (at runtime) using a form of memory allocation known as **dynamic memory allocation****. This ability to acquire additional memory is part of what **makes `std::string` so flexible, but also comparatively slow**.

We cover dynamic memory allocation in a future chapter.

### 7.9.2 String input with `std::cin`

Using `std::string` with `std::cin` may yield some surprises! Consider the following example:

```

4 int main()
5 {
6     std::cout << "Enter your full name: ";
7     std::string name{};
8     std::cin >> name; // this won't work as expected since std::cin breaks on whitespace
9
10    std::cout << "Enter your favorite color: ";
11    std::string color{};
12    std::cin >> color;
13
14    std::cout << "Your name is " << name << " and your favorite color is " << color << '\n';
15
16    return 0;
17 }

```

Output:

Enter your full name: John Doe

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

It turns out that when using operator<> to extract a string from `std::cin`, **operator<> only returns characters up to the first whitespace it encounters**. Any **other characters are left inside `std::cin`**, waiting for the next extraction.

So, when we used operator<> to extract input into variable `name`, only "John" was extracted, leaving " Doe" inside `std::cin`. When we then used operator<> to get extract input into variable `color`, it extracted "Doe" instead of waiting for us to input a color. Then the program ends.

### 7.9.3 Use `std::getline()` to input text

To read a full line of input into a string, you're better off using the `std::getline()` function instead. `std::getline()` requires two arguments: the first is `std::cin`, and the second is your string variable.

```

1 #include <iostream>
2 #include <string> // For std::string and std::getline
3
4 int main()
5 {
6     std::cout << "Enter your full name: ";
7     std::string name{};
8     std::getline(std::cin >> std::ws, name); // read a full line of text into name
9

```

#### 7.9.4 *What is std::ws? (input manipulator)*

In Floating point numbers, **we discussed output manipulators**, which allow us to **alter the way output is displayed**. In that lesson, we used the output manipulator function **std::setprecision()** to change the number of digits of precision that **std::cout** displayed.

**C++ also supports input manipulators**, which **alter the way that input is accepted**. The **std::ws** input manipulator tells **std::cin** to **ignore any leading whitespace before extraction**. Leading whitespace is any whitespace character (spaces, tabs, newlines) that occur at the start of the string.

##### **Best practice:**

If using **std::getline()** to read strings, **use std::cin >> std::ws** input manipulator to **ignore leading whitespace**. This **needs to be done for each std::getline() call**, as **std::ws** is not preserved across calls.

##### **Key insight:**

When extracting to a variable, the **extraction operator (>>)** **ignores leading whitespace**. It **stops extracting when encountering non-leading whitespace**.

**std::getline()** **does not ignore leading whitespace**. If you want it to ignore leading whitespace, **pass std::cin >> std::ws as the first argument**. It stops extracting when encountering a newline.

### 7.9.5 The length of a `std::string`

If we want to know how many characters are in a `std::string`, we can ask a `std::string` object for its length. The syntax for doing this is different than you've seen before, but is pretty straightforward:

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string name{ "Alex" };
7      std::cout << name << " has " << name.length() << " characters\n";
8
9      return 0;
10 }
```

Although `std::string` is required to be null-terminated (as of C++11), the **returned length of a `std::string` does not include the implicit null-terminator** character.

**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.

Key insight:

With **normal functions**, we call `function(object)`. With **member functions**, we call `object.function()`.

**Also note that `std::string::length()` returns an unsigned integral value** (most likely of type `size_t`). **If you want to assign the length to an int variable**, you should `static_cast` it to avoid compiler warnings about signed/unsigned conversions:

```
1 | int length { static_cast<int>(name.length()) };
```

In C++20, you can also use the `std::ssize()` function to get the length of a `std::string` as a large **signed integral type** (usually `std::ptrdiff_t`). Since a `ptrdiff_t` may be larger than an `int`, if you want to store the result of `std::ssize()` in an `int` variable, **you should `static_cast` the result to an `int`**.

### 7.9.6 *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.

### 7.9.7 *Do not pass std::string by value*

When a `std::string` is passed to a function by value, the `std::string` function parameter must be instantiated and initialized with the argument. This **results in an expensive copy**.

**Best practice:**

Do not pass `std::string` by value, as it makes an expensive copy. **In most cases, use a `std::string_view` parameter instead.**

### 7.9.8 *Returning a std::string*

In C++, when a function returns a value, you might be concerned that returning a `std::string` by value could be expensive because it would involve copying the string. However, modern C++ compilers optimize this process using techniques like **Return Value Optimization (RVO)** and **move semantics**, making it efficient to return `std::string` by value in many **common scenarios**.

As a **rule of thumb**, it is okay to return a `std::string` by value in following scenarios:

- A **local (to the function)** variable of type `std::string`. (**RVO**)
- A `std::string` returned by another function call or operator (**move semantics**).
- A `std::string` temporary that is created as part of the return statement. (**RVO or move semantics**)

**For advanced readers:**

`std::string` supports a capability called **move semantics**, which allows an object that will be destroyed at the end of the function to instead be returned by value without making a copy.

How move semantics works is beyond the scope of this introductory article, but is something we introduce in lesson: Returning `std::vector`, and an introduction to move semantics. **In most other cases, prefer to avoid returning a `std::string` by value**, as doing so will make an expensive copy.

### 7.9.9 Literals for `std::string`

Double-quoted string literals (like “Hello, world!”) are C-style strings by default (and thus, have a strange type).

We can create string literals with type `std::string` by using a **s suffix** after the double-quoted string literal. The s must be lower case.

```
1  #include <iostream>
2  #include <string> // for std::string
3
4  int main()
5  {
6      using namespace std::string_literals; // easy access to the s suffix
7
8      std::cout << "foo\n"; // no suffix is a C-style string literal
9      std::cout << "goo\n"s; // s suffix is a std::string literal
10
11     return 0;
12 }
```

**Tip:** The “s” suffix lives in the `namespace std::literals::string_literals`.

**The most concise way to access the literal suffixes is via using-directive using namespace `std::literals`.** However, this imports all of the standard library literals into the scope of the using-directive, which brings in a bunch of stuff you probably aren’t going to use.

**We recommend using namespace `std::string_literals`, which imports only the literals for `std::string`.**

We discuss using-directives in the future. This is one of the exception cases where using an entire namespace is generally okay, because the suffixes defined within are unlikely to collide with any of your code. **Avoid such using-directives outside of functions in header files.**

You probably won’t need to use `std::string` literals very often (**as it’s fine to initialize a `std::string` object with a C-style string literal**), but we’ll see a few cases in future lessons (involving type deduction) where using `std::string` literals instead of C-style string literals makes things easier.

**For advanced readers:**

`"Hello"s` resolves to `std::string { "Hello", 5 }` which creates a temporary `std::string` initialized with C-style string literal “Hello” (which has a length of 5, excluding the implicit null-terminator).

### 7.9.10 *Constexpr strings*

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**.

### 7.9.11 *Conclusion*

**`std::string` is complex**, leveraging many language features that we haven't covered yet. Fortunately, you don't need to understand these complexities to use `std::string` for simple tasks, like basic string input and output. **We encourage you to start experimenting with strings now, and we'll cover additional string capabilities later.**

## 7.10 Introduction to `std::string_view`

Consider the following program:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 5 }; // x makes a copy of its initializer
6     std::cout << x << '\n';
7
8     return 0;
9 }
```

When the definition for `x` is executed, the initialization value 5 is copied into the memory allocated for variable `int x`. **For fundamental types, initializing and copying a variable is fast.**

Now consider this similar program:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string s{ "Hello, world!" }; // s makes a copy of its initializer
7     std::cout << s << '\n';
8
9     return 0;
10 }
```

When `s` is initialized, the C-style string literal "Hello, world!" is copied into memory allocated for `std::string s`. **Unlike fundamental types, initializing and copying a `std::string` is slow.**

In the above program, all we do with `s` is print the value to the console, and then `s` is destroyed. **We've essentially made a copy of "Hello, world!" just to print and then destroy that copy. That's inefficient.**



We see something similar in this example:

```
1 #include <iostream>
2 #include <string>
3
4 void printString(std::string str) // str makes a copy of its initializer
5 {
6     std::cout << str << '\n';
7 }
8
9 int main()
10 {
11     std::string s{ "Hello, world!" }; // s makes a copy of its initializer
12     printString(s);
13
14     return 0;
15 }
```

This example makes two copies of the C-style string “Hello, world!”: one when we initialize `s` in `main()`, and another when we initialize parameter `str` in `printString()`. That’s a lot of needless copying just to print a string!

### 7.10.1 `std::string_view`

To address the issue with `std::string` being expensive to initialize (or copy), C++17 introduced `std::string_view` (which lives in the `<string_view>` header). `std::string_view` provides read-only access to an existing string (C-style, `std::string`, or `std::string_view`) without making a copy. Read-only means that we can access and use the value being viewed, but we cannot modify it.

The following example is identical to the prior one, except we’ve replaced `std::string` with `std::string_view`. **Read-only** means that we can access and use the value being viewed, but we cannot modify it.

The following example is identical to the prior one, except we've replaced `std::string` with `std::string_view`:

```
1  #include <iostream>
2  #include <string_view> // C++17
3
4  // str provides read-only access to whatever argument is passed in
5  void printSV(std::string_view str) // now a std::string_view
6  {
7      std::cout << str << '\n';
8  }
9
10 int main()
11 {
12     std::string_view s{ "Hello, world!" }; // now a std::string_view
13     printSV(s);
14
15     return 0;
16 }
```

This program produces the same output as the prior one, but no copies of the string “Hello, world!” are made.

When we initialize `std::string_view s` with C-style string literal “Hello, world!”, `s` provides read-only access to “Hello, world!” without making a copy of the string. **When we pass `s` to `printSV()`, parameter `str` is initialized from `s`. This allows us to access “Hello, world!” through `str`, again without making a copy of the string.**

#### Best practice:

Prefer **`std::string_view`** over `std::string` when you need a read-only string, especially for function parameters.

### 7.10.2 *`std::string_view` can be initialized with many different types of strings*

One of the neat things about a `std::string_view` is how flexible it is. A `std::string_view` object can be initialized with:

- a C-style string
- a `std::string`
- a `std::string_view`

### 7.10.3 *std::string\_view parameters will accept many different types of string arguments*

Both a C-style string and a `std::string` will implicitly convert to a `std::string_view`. Therefore, a `std::string_view` parameter will accept arguments of type C-style string, a `std::string`, or `std::string_view`.

### 7.10.4 *std::string\_view will not implicitly convert to std::string*

Because `std::string` makes a copy of its initializer (which is expensive), C++ won't allow implicit conversion of a `std::string_view` to a `std::string`. This is to prevent accidentally passing a `std::string_view` argument to a `std::string` parameter, and inadvertently making an expensive copy where such a copy may not be required.

However, if this is desired, we have two options:

- Explicitly create a `std::string` with a `std::string_view` initializer (which is allowed, since this will rarely be done unintentionally)
- Convert an existing `std::string_view` to a `std::string` using `static_cast`

## 7.11 `std::string_view` (part 2)

## 7.12 Chapter 5 summary and quiz