Summary: C++

AgC86

-

Learncpp

Author Note

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

1. Contents

C++

1 Chapter 0: Introduction / Getting started

1.1 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. 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.

1.1.1 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**.

1.1.2 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.

1.1.3 High-level languages

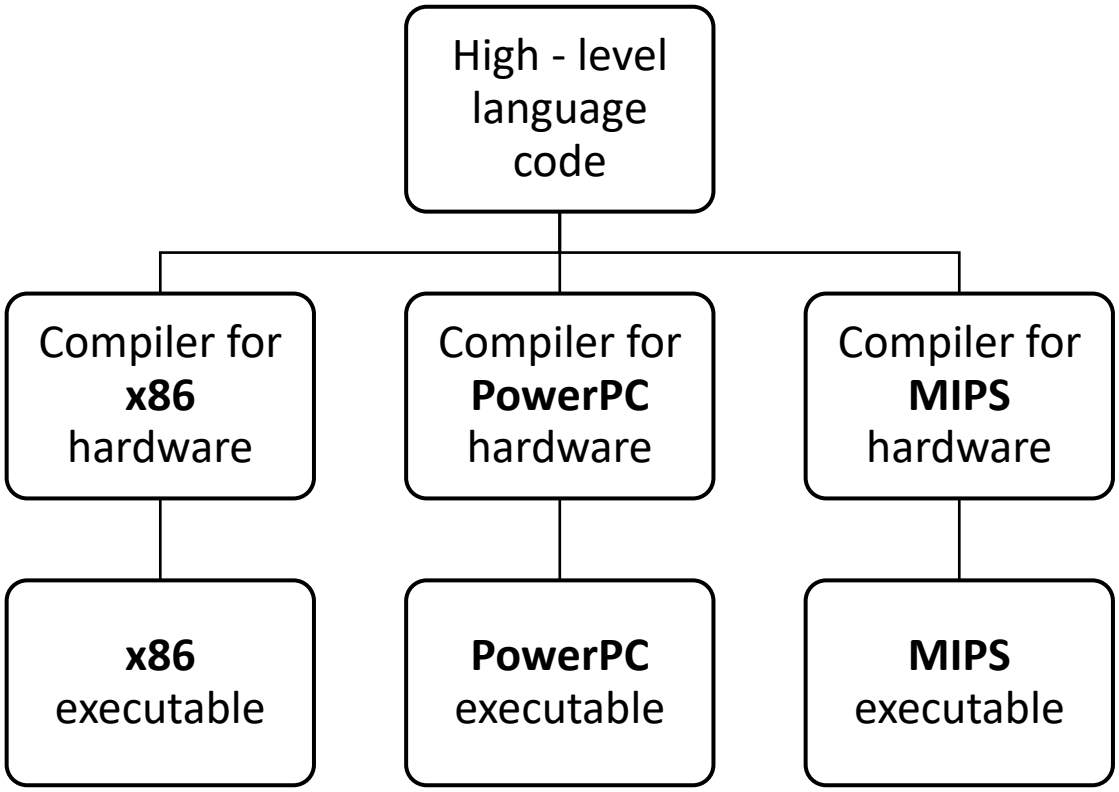
Now that we know some of the downsides:

- 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.

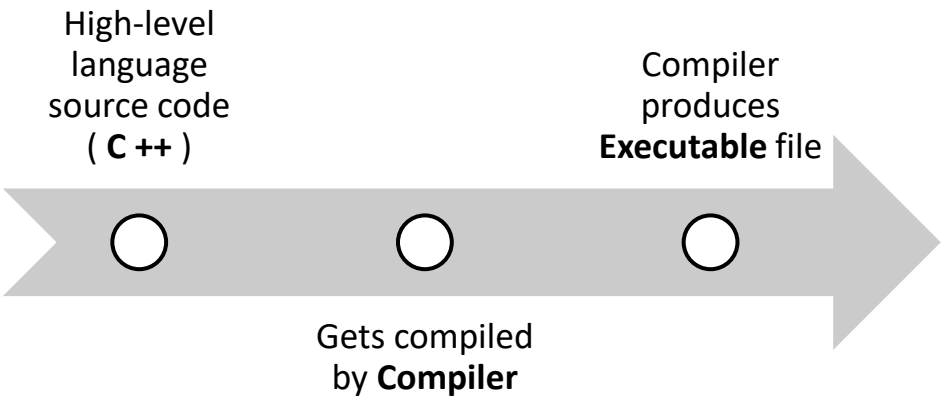
Here is the same instruction as above in C/C++: **a = 97;**

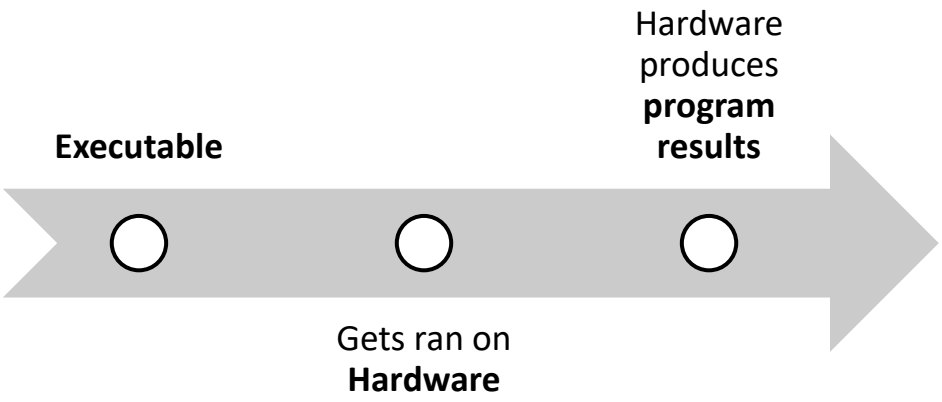
Much like assembly programs, programs written in high level languages must be translated into a format the computer can understand before they can be run. There are two primary ways this is done: compiling and interpreting. Most languages can be compiled or interpreted. Traditionally languages like **C, 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).


High - level language code
Compiler for **x86** hardware
Compiler for **PowerPC** hardware
Compiler for **MIPS** hardware
x86 executable
PowerPC executable
MIPS executable

1.1.3.1 Compiling

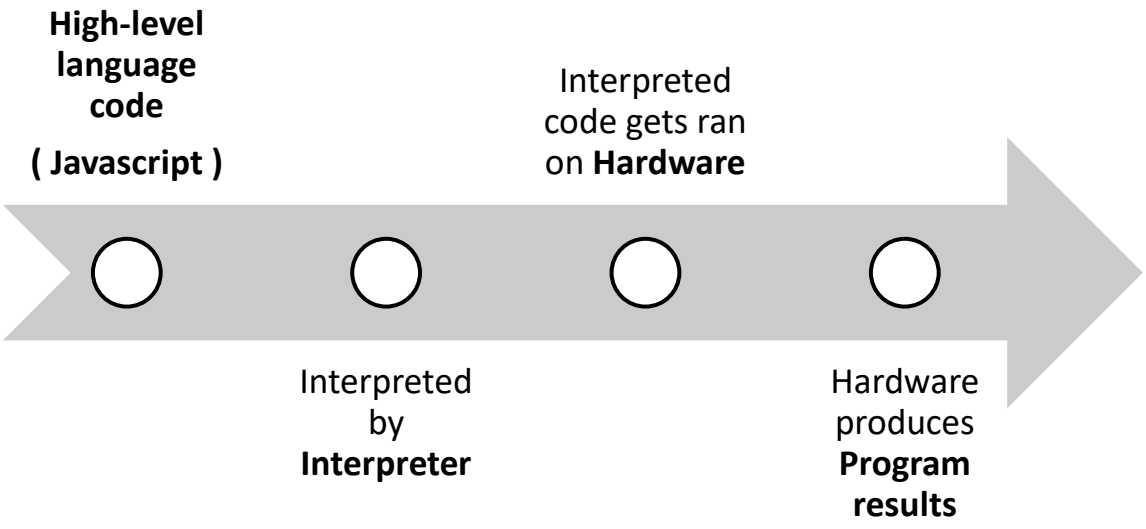
A **compiler** is a program (or collection of programs) that reads source code (typically written in a high-level language) and **translates** it **into** some other language (typically a **low-level language**, such as assembly or machine language, etc.). Most often, these low-level language files are 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**).





1.1.3.2 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.**



1.2 Introduction to C / C++

1.2.1 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)

1.2.1.1 Brief history: C

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

1.2.1.1.1 1973: Unix

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

1.2.1.1.2 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**.

1.2.1.1.3 1983: Forming of ANSI committee

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

1.2.1.1.4 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.

1.2.1.1.5 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.

1.2.1.1.6 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++.

1.2.1.1.7 Later releases

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

1.2.1.2 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

1.2.1.2.1 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)

1.2.1.2.2 2003: Release: C++03 standard

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

1.2.1.2.3 Later releases

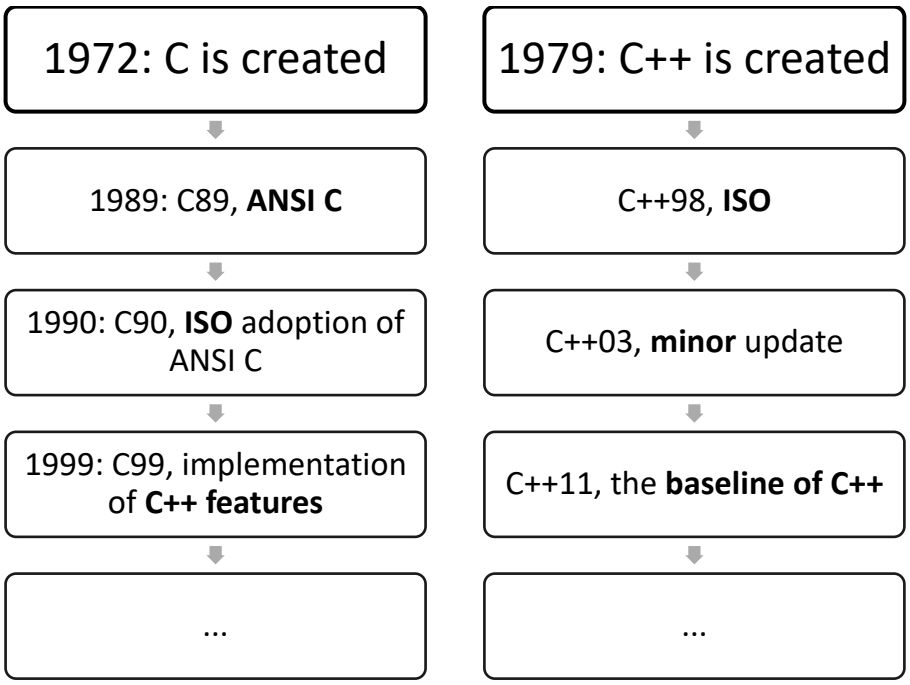
Five major updates to the C++ language (**C++11, C++14, C++17, C++20, and C++23**) have been made since then, each adding additional functionality. **C++11 in particular added a huge number of new capabilities, and is widely considered to be the new baseline version of the language.**

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.

1.2.1.3 Summary of language standards

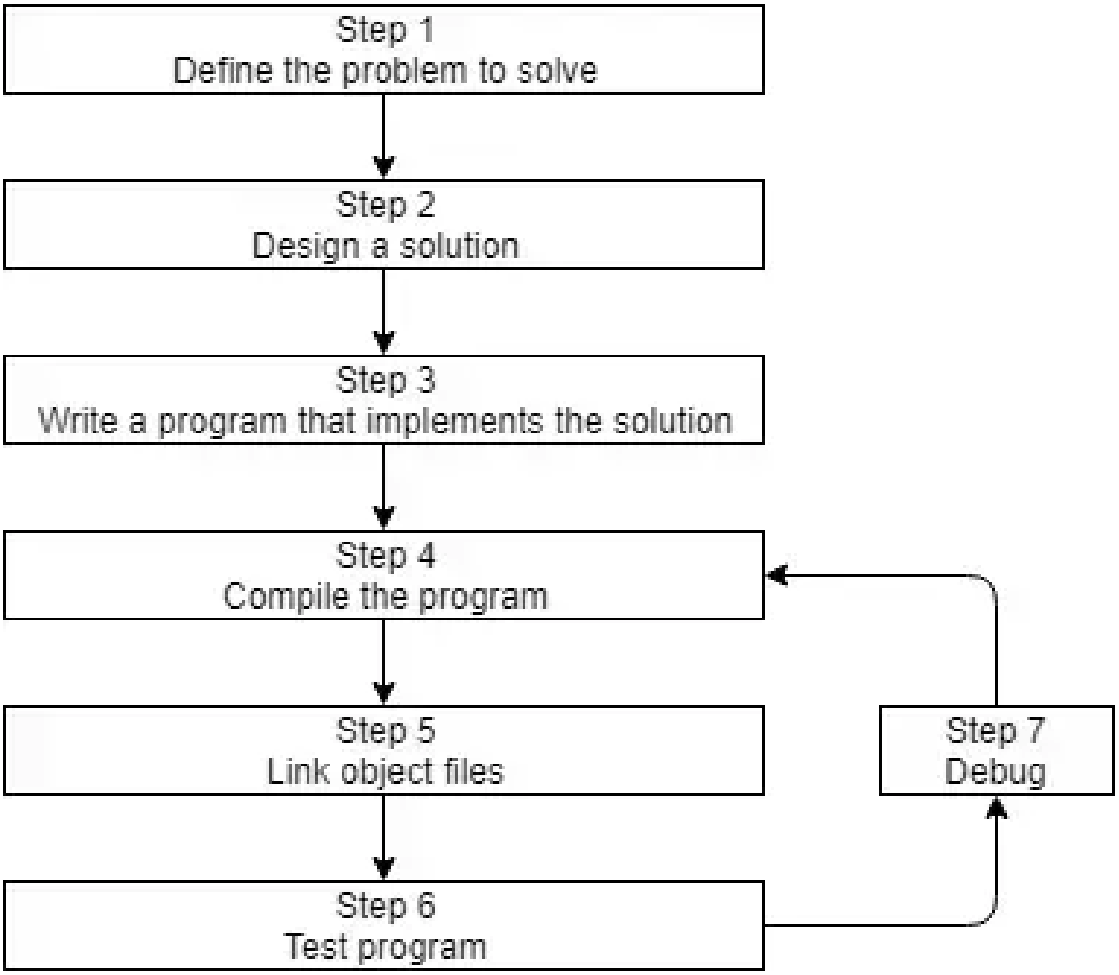


1.2.2 C / C++ philosophy

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.**

1.3 Introduction to C++ development

1.3.1 C++ development flow



1.4 Introduction to the compiler, linker and libraries

1.4.1 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**.

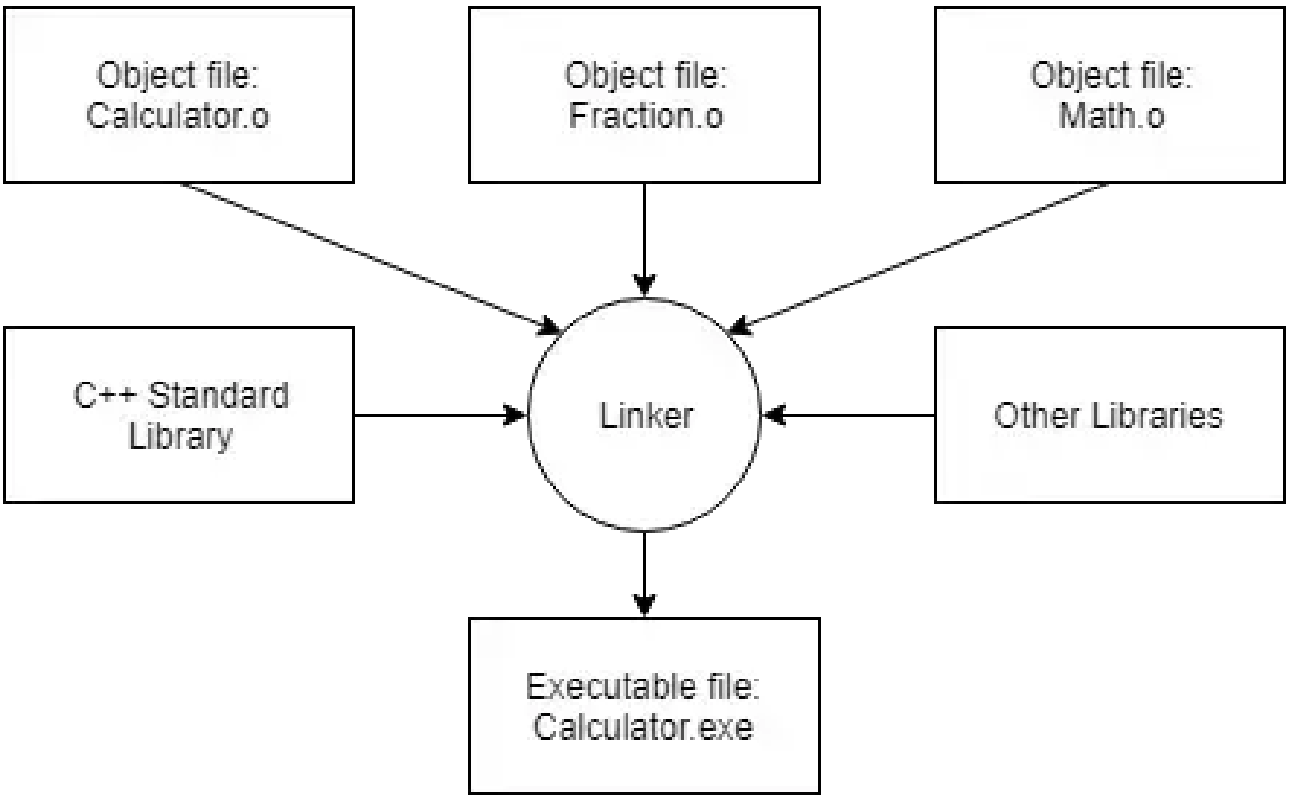
Example**: Calulator.cpp get compiled into Calculator.o**

1.4.2 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.


Object file:
Calculator.o
Object file:
Fraction.o
Object file:
Math.o
C++ Standard
Library
Linker
Other Libraries
Executable file:
Calculator.exe

1.4.3 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**.

1.4.3.1 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.

1.5 Configuring your compiler

1.5.1 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.

1.5.1.1 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.

1.5.1.2 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).

1.5.2 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).

1.5.3 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).

1.5.3.1 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.

1.5.3.2 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.

1.5.3.3 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.

1.5.3.4 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.

1.5.4 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.

1.5.4.1 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).

1.5.4.2 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.

2 Chapter 1: Basics

2.1 Statements and the structure of a program

2.1.1 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.

2.1.1.1 Different kinds of statements

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

2.1.2 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).

2.1.3 Identifiers

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

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.3 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.

2.3.1 Key Insight

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

2.3.2 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”).

2.3.2.1 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

2.3.3 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.**

2.3.3.1 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.

2.3.3.2 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.**

2.3.3.3 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**.

2.3.4 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.

2.3.4.1 Defining a variable of a certain type

```
Int a;              ← This is a definition
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**).

2.4 Variable assignment

2.4.1 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).

2.4.2 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
```

2.5 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**

2.5.1 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)
```

2.5.1.1 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.**

2.5.1.2 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.**

2.5.1.3 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.

2.5.1.4 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:

Int a {}; is the same as writing int a = 0;

2.5.1.5 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.**

2.6 Introduction to iostream

2.6.1 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
```

2.6.2 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;
```

2.6.2.1 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).

2.6.3 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;
```

Result:

Hi!

My name is Alex.

2.6.3.1 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.

2.6.3.2 Best practice

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

2.6.4 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';
```

2.6.4.1 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.**

2.7 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**.

2.7.1 Nomenclature

Many readers expect the terms “initialized” and “uninitialized” to be strict opposites, but they aren’t quite! 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.**

2.7.2 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.

2.7.3 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.

2.7.4 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.

2.7.5 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:

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

2.7.6 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.**

2.7.7 Best practice

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

2.8 Keywords and naming identifiers

2.8.1 Keywords

A - C	D - P	R - Z
alignas (C++11)	decltype (C++11)	reflexpr (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 (TM TS)	dynamic_cast	short
atomic_commit (TM TS)	else	signed
atomic_noexcept (TM TS)	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 (TM TS)
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
consteval (C++20)	noexcept (C++11)	union
constexpr (C++11)	not	unsigned
constinit (C++20)	not_eq	using (1)
const_cast	nullptr (C++11)	virtual
continue	operator (4)	void
co_await (C++20)	or	volatile
co_return (C++20)	or_eq	wchar_t
co_yield (C++20)	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.

2.8.2 Identifiers

2.8.2.1 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

2.8.2.2 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.

2.9 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

2.9.1.1 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

```cpp
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
E0065	expected a ';'
C2017	illegal escape sequence
E0008	missing closing quote
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'
E0007	unrecognized token

← **Error output**

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

    return 0;
}
```

← **Correct newline implementation**

2.9.1.2 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

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

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

```
#include <iostream>

int main()
{
    std::cout << "Hello world";

    return 0;
}
```

2.9.1.3 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.**

2.9.1.4 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).
    a. 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.
    b. 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
    a. **K & R** or **C-style**, puts the opening bracket on the same line as the statement
        i. **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;
}
```

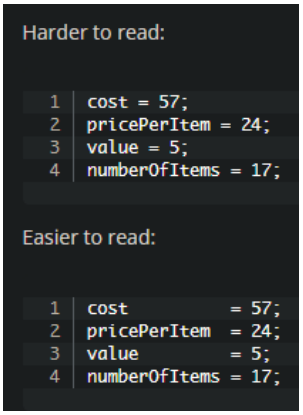
    b. **Allman**, put the opening bracket on a separate line right under the statement
        i. This **enhances readability**, and is **less error prone** since your brace pairs should always be indented at the same level. If you get a compiler error due to a brace mismatch, it's very easy to see where.

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

    return 0;
}
```

3. Each statement within curly braces should start one tab in from the opening brace of the function it belongs to
4. **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.
5. 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.



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

2.9.1.5 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.

2.10 Introduction to literals and operators

2.11 Introduction to expressions

2.12 Developing your first program