C++ Complete Roadmap

Basic Definitions

A **computer program** (also commonly called an **application**) is a set of instructions that the computer can perform in order to perform some task.

Programming is the process of creating a program which is done by producing(or writing) **source code** shortened to **"code"** (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.

A **compiler** is a program that reads source code and produces a stand-alone executable program that can then be run. Once your code has been turned into an executable, you do not need the compiler to run the program.

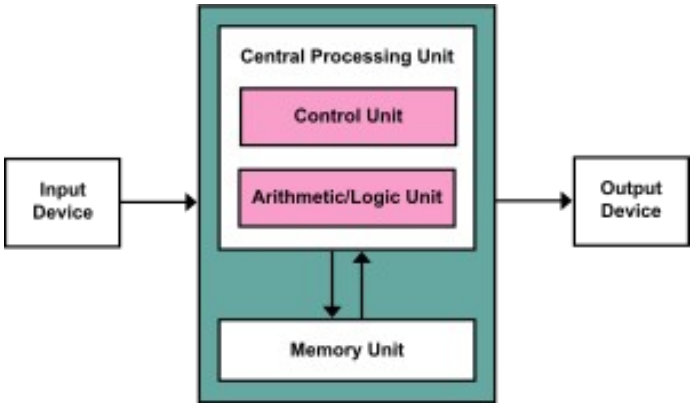
An **interpreter** is a program that directly executes the instructions in the source code without requiring them to be compiled into an executable

---

Von Neumann Architecture

Historically there have been two types of computers; Fixed Program Computers with very specific purpose and could not be reprogramed like calculators and Stored Program Computers which can carry out many different tasks as applications are stored on them

Von Neuman Architecture is a Stored Program Concept that is used in modern computers which has a Central Processing Unit with a Arithmetic Logical Unit to perform Mathematical & Logical Operations and a Control unit that directs all I/O flow, interprets instructions and controls movements of data around the system



Memory Units

The smallest unit of memory in a computer is called **bit** and is either a 0 or 1(the two; off and on states of a transistor based on small base current flowing or not)

whereas **bytes** are collection of 8 bits.

8 bits = 1 byte

1024 Bits = 1 Kilobit (Kb)	1024 Bytes = 1 Kilobyte (KB)
1024 Kb = 1 Megabit (Mb)	1024 Kilobytes = 1 Megabyte (MB)
1024 Mb = 1 Gigabit (Gb)	1024 Megabytes = 1 Gigabyte (GB)

We use the Byte system in modern terms to refer to data (It's a 5 GB movie file) however internet speed tests usually give the result in bits for more accuracy as that's how data is transmitted (the speed is 50Mbps).

Memory Storage Unit

Primary Memory (Smaller Storage / Not Accessible by User / Faster Speed)

Registers are fastest high speed memory unit that store the data which needs to be processed by the CPU. They usually hold 32 - 64 bits of data.

Cache Memory are high speed memory located between CPU and Main Memory store data & instructions that need to be executed. They usually hold 1- 8 Megabytes of data.

RAM or Random Access Memory is a **Volatile Memory Unit** that is roughly 10-100 times slower than cache memory and is a Read & Write Memory that stores information typed by the user and can be accessed directly without sequential scanning of data as it has the property of being randomly accessed. They usually hold 2 - 32 Gigabytes of data.

ROM or Read only Memory which is a **Non-Volatile Memory Unit** which won't lose its contents even after power is switched off and stores essential boot-up instructions, major I/O tasks and software instructions and usually hold 4 – 8 Megabytes of data.

Secondary Memory (Larger Storage / Accessible by User / Slower Speed)

Solid State Storage are permanent semiconductor chips based memory units including Flash Storage & Solid State Drives and are used to store portable user data.

Magnetic Storage are magnetic field based memory units that include Hard Disk Drives and Magnetic Tapes. They comparatively slower then Solid State Storage & are also used to store portable user data.

Machine Language

A computer's CPU is incapable of speaking C++. The limited set of instructions that a CPU can understand directly is called **machine code** (or **machine language** or an **instruction set**) which looked like `10110000 01100001`

Back when computers were first invented, programmers had to write programs directly in machine language, which was a very difficult and time consuming thing to do.

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 which looked like `mov al, 061h`

However, The CPU still can't read assembly language and an **Assembler** is used to convert the Assembly Language into Machine Language.

High Level Languages

Languages that have high ease of readability and are programmer friendly to use are known as high level languages. They are far different looking from machine or processor language.

However they are comparatively slower to run than Low-level Languages like Machine or Assembly language. This is why Assembly Language is still used when speed is critical.

Languages like Java and Python are considered High-level Languages while C/C++ are called Mid-level languages.

Integrated Development Environment

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

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

Examples are Visual Studio Code, Code::Blocks, Xcode.

Before C++, there was C

The C language was developed in 1972 by Dennis Ritchie at Bell Telephone laboratories, primarily as a systems programming language (a language to write operating systems with). C ended up being so efficient and flexible that in 1973, Ritchie and Ken Thompson rewrote most of the Unix operating system using C. In 1983, the American National Standards Institute (ANSI) formed a committee to establish a formal standard for C. In 1989 (committees take forever to do anything), they finished, and released the C89 standard, more commonly known as ANSI C.

C++ Programming Language

C++ (pronounced “see plus plus”) was developed by Bjarne Stroustrup at Bell Labs as an extension to C, starting in 1979. C++ adds many new features to the C language, and is perhaps best thought of as a superset of C, , though this is not strictly true (as C99 introduced a few features that do not exist in C++). The claim to fame for C++ results primarily from the fact that it is an object-oriented language.

C and C++’s 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.

C++ Standard Library

It is an extensive library that provides a set of useful capabilities for use in your programs. One of the most commonly used parts of the C++ standard library is the #include <iostream> which contains functionality for printing text on a monitor and getting keyboard input from a user, from implemented data types like strings & vectors(STL) to functions like std::sort are all part of Standard Library & can be used after the part of SL included is like #include <iostream>, #include <algorithm>, #include <vector>.

Statement

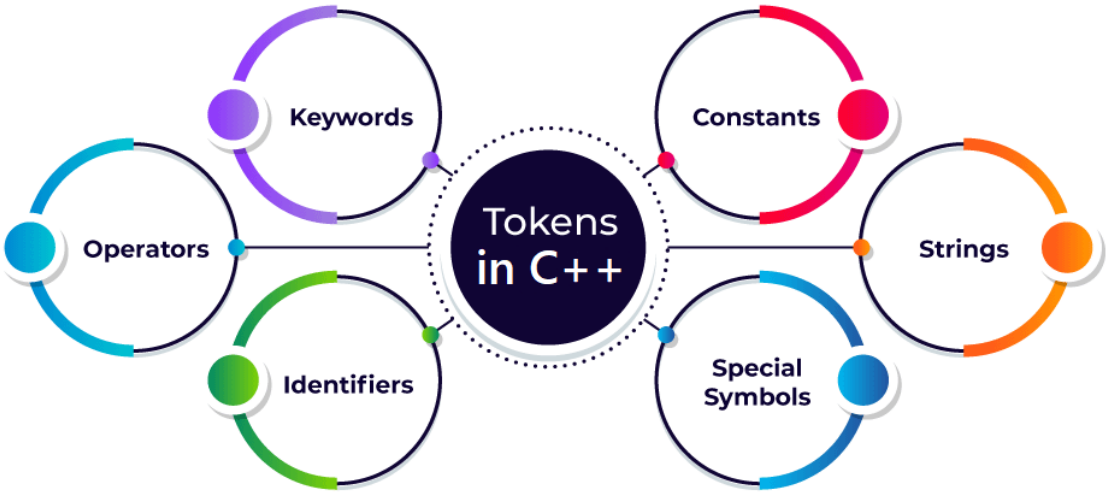
A computer program is a sequence of instructions that tell the computer what to do.
A **statement** is a type of instruction that causes the program to *perform some action*.

Functions

A function is a group of statements between curly braces that get executed sequentially. Every C++ program must have a special function called main()

tokens

tokens are smallest individual units in the source code which are recognized by the compiler and include various subtypes like.



keywords

C++ reserves a set of 92 words (as of C++23) for its own use. These words are called **keywords** (or reserved words), and each of these keywords has a special meaning within the C++ language.

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

comments

a **comment** is a programmer-readable note that is inserted directly into the source code of the program. Comments are ignored by the compiler and are for the programmer's use only.

you can have single line comments with `//` like

```
std::cout << "Hello world!\n"; // std::cout lives in the iostream library
std::cout << "It is very nice to meet you!\n"; // these comments make the code hard to read
std::cout << "Yeah!\n"; // especially when lines are different lengths
```

or multi line comments with `/*` and `*/` pair like

```
/* This is a multi-line comment.
   This line will be ignored.
   So will this one. */
```

it is good practice to comment your code liberally as if you are talking to somone who has no idea about the code and its wrong to assume you will remember why you made specific choices regarding the code in the future.

data and values

data is any information that can be moved, processed, or stored by a computer. Programs produce results by reading writing and changing data.
a single piece of data is called a **value** (examples - a, 5 and Hello)

when we run a program the operating system loads it into the RAM in binary as that is where user types information goes in. RAM can be imagined as a series of numbered boxes that store data. In old programming languages, you could even access these boxes.

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.
an **object** is used to store a value in memory. A **variable** is an object that has a name (identifier).

data types

a **data type** (more commonly just called a **type**) tells the compiler what type of value (e.g. a number, a letter, text, etc...) the variable will store. They include char(characters), strings, int (integers), short(short integers), floating point numbers and doubles.

declaring Variables

```
1 | int x; // define a variable named x, of type int
```

defining Multiple Variables

you can either use

Int a;
Int b;

or

int a, b;

however you must avoid common declaration errors like

```
1 | int a, int b; // wrong (compiler error)
2 |
3 | int a, b; // correct
```

```
int a, double b; // wrong (compiler error)

int a; double b; // correct (but not recommended)

// correct and recommended (easier to read)
int a;
double b;
```

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**. 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.

```
int width; // define an integer variable named width
width = 5; // assignment of value 5 into variable width
```

the variable can change values after sequential assignment so for

int variable:
variable = 10;
variable = 11;

and if we print variable we will only get 11.

variable initialization

assignment requires minimum two statements to assign value to variable whereas we can use initialization to give value to variable with one line only

```
int a;         // no initializer (default initialization)
int b = 5;     // initializer after equals sign (copy initialization)
int c( 6 );    // initializer in parenthesis (direct initialization)
```

copy initialization was inherited from C but has fallen out of favour due to low efficiency

direct initialization has also fallen out of favour as it can be mistaken for a function

```
int x();  // forward declaration of function x
int x(0); // definition of variable x with initializer 0
```

list initialization is the modern way to initialize objects in C++

```
int width { 5 };    // direct list initialization of value 5 into variable width
int height = { 6 }; // copy list initialization of value 6 into variable height
int depth {};       // value initialization (see next section)
```

direct list & value initialization is the most preferred way to initialize variables in C++

value initialization initializes the value to zero (or empty for other datatypes) and is used for variables that will be immediately given a value soon.

initialization multiple values

```
int a = 5, b = 6;           // copy initialization
int c( 7 ), d( 8 );         // direct initialization
int e { 9 }, f { 10 };      // direct brace initialization
int g = { 9 }, h = { 10 }; // copy brace initialization
int i {}, j {};             // value initialization
```

direct brace initialization is preferred way for assigning multiple values.

unused initialized variables warning

modern compilers will typically generate warnings if variables are initialized but not used and its better to not use such variables as these warnings could be upgraded to “errors” for the compiler.

however if we must use such variables like constants that may or may not be used, we can use **[[maybe_unused]]** before datatype can be used to tell the compiler we are okay with unused variables

```
int main()
{
    double pi { 3.14159 };
    double gravity { 9.8 };
    double phi { 1.61803 };

    // assume some of the above are used here, some are not

    return 0;
}
```

whitespace independency

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.

The following variable definitions are all valid:

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

the input/output library

The input/output library (**io library**) is part of the C++ standard library that deals with basic input and output. It has the functionality to get input from keyboard and output to console. It includes stream objects like **std::cout** & **std::endl**

#include<iostream> //header file is used to activate the functionality

std::cout

it allows us to send data to the console as printed text. It stands for “character output”.

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

int main()
{
    std::cout << "Hello world!"; // print Hello world! to console

    return 0;
}
```

also can be used to

print value of variables

int x{5};

std::cout << x; //prints 5 to the console

concatenate multiple pieces
std::cout << “Hello” << “world!” ;

or with **variable values**
int x{5};
std::cout << “x is equal to: “ << x; //will print – **x is equal to : 5**

std::endl

Since separate output statements don’t lead to separate lines of output on the console so

std::cout << “hi”;
std::cout << “my name is varun”;

/*will print it as
himy name is alex*/

one way to fix this is by std::endl

std::cout <<”hi” << std::endl;
std::cout <<”my name is varun” << std::endl;

will give us both results on different lines

std::endl vs ‘\n’

---

using std::endl is inefficient as it not only moves the program to next line but also flushes the buffer (removal of all the output queues that are not immediately displayed in console) which is not needed as many times and is done periodically by the system design itself.

std::cout << “x is equal to” << x << ‘\n’; // \n is between single quotes when used alone

std::cout <<”hello varun’s friend \n”;

\n is preffered over std::endl when outputting text to console

std::cin

character input is used to read input from keyboard

```
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
```

you can also have **multiple inputs in a single text of line**

```cpp
int x{}; // define variable x to hold user input (and value-initialize it)
int y{}; // define variable y to hold user input (and value-initialize it)
std::cin >> x >> y; // get two numbers and store in variable x and y respectively

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

advanced I/O

https://www.learncpp.com/cpp-tutorial/input-and-output-io-streams/

https://www.learncpp.com/cpp-tutorial/input-with-istream/

https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/

https://www.learncpp.com/cpp-tutorial/stream-classes-for-strings/

https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/

https://www.learncpp.com/cpp-tutorial/basic-file-io/

https://www.learncpp.com/cpp-tutorial/random-file-io/

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 (for example cout could be a function/identifier of different meaning but when we write std::cout, it means the cout function that is included in std namespace where :: is scope resolution operator)

we can also use directive statements for declaring namespace scope but this is highly discourages especially when used in global namespace or other namespaces because future language updates can make keywords collide with their naming as there are hundreds if not thousands of common names in std namespace itself like count, min, max, search, sort. This also violates the reason , namespaces were created in the first place which was to have separate uses for same keyword. The scope for using statements, if used inside a code block, ends with the block. **using declarations** are in general safer than **using directives.**

using directives

```cpp
#include <iostream>

using namespace std; // this is a using-directive that allows us to access names in the std namespace with no namespace prefix

int main()
{
    cout << "Hello world!";
    return 0;
}
```

using declarations

```cpp
#include <iostream>

int main()
{
   using std::cout; // this using declaration tells the compiler that cout should resolve to std::cout
   cout << "Hello world!\n"; // so no std:: prefix is needed here!

   return 0;
} // the using declaration expires at the end of the current scope
```

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**).

```cpp
// All of the following statements are part of the global namespace

void foo();    // okay: function forward declaration
int x;         // compiles but strongly discouraged: non-const global variable definition (without initializer)
int y { 5 };   // compiles but strongly discouraged: non-const global variable definition (with initializer)
x = 5;         // compile error: executable statements are not allowed in namespaces

int main()     // okay: function definition
{
    return 0;
}

void goo();    // okay: A function forward declaration
```

basic formatting

the Google C++ style guide recommends putting the opening curly braces on the same line as the statement

```cpp
int main() {
}
```

however a more readable and less error prone would be having curly braces indented on same level to debug brace mismatch easily

```
int main()
{
}
```

each statement within curly braces should start one tab in from the opening brace of the function it belongs to.

```
int main()
{
    std::cout << "Hello world!\n"; // tabbed in one tab (4 spaces)
    std::cout << "Nice to meet you.\n"; // tabbed in one tab (4 spaces)
}
```

when a statement with many operations must be written, the operator must be written first

```
std::cout << 3 + 4
    + 5 + 6
    * 7 * 8;
```

some more examples of easy to read code is

```
cost          = 57;
pricePerItem  = 24;
value         = 5;
numberOfItems = 17;
```

```
std::cout << "Hello world!\n";                // cout lives in the iostream library
std::cout << "It is very nice to meet you!\n"; // these comments are easier to read
std::cout << "Yeah!\n";                       // especially when all lined up
```

functions

A **function** is a reusable sequence of statements designed to do a particular job. every executable program must have a function named *main* (which is where the program starts execution when it is run). C++ standard library comes with plenty of already-written functions. Functions that you write yourself are called **user-defined functions**. One should use functions to follow through one of the central tenets of good programming which is **don't repeat yourself (DRY).**

```
returnType functionName()   //function header
{
```

```
        //function body
}
```

for example lets say

```
#include <iostream>

void doPrint()
{
        std::cout << “How are you doing?”;
}

Int main()
{
        doPrint();
        return 0;
}

/*will print the output

How are you doing? */
```

we can call function inside of functions so so lets say doPrintA() is a function and doPrintB() is another function then we can call doPrintB() inside doPrintA()

nested functions aren’t allowed though which means that declaring functions inside of main() is not allowed in C++ & all functions must be declared outside of main()

we can also use forward declaration to make functions written after main() work by writing the returnType functionName(parameters); before int main(){}

using value-returning functions for variables

```cpp
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value the user entered back to the caller
}

int main()
{
    int num { getValueFromUser() }; // initialize num with the return value of getValueFromUser()

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

a function that returns value (that is not void()) must have a return statement at the end of it or there may be undefined behaviour and compiler can be unable to determine the error.

a value-returning function can only return one value

working of main() function

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. The return value from `main` is sometimes called a **status code** (also sometimes called an **exit code**, or rarely a **return code**), as it is used to indicate whether the program ran successfully or not.
by definition, a status code of `0` means the program executed successfully. main() will return 0 if not return statement is provided in source code even if its best practice to mention it.

C++ disallows calling of main function explicitly so a main() inside of int main() may lead to compiler error even though this can be used in C for repeating the int main() function again and again.

working of void() function

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

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

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

functions including print statements don't need to return any value however these statements can't be used to return any values as the name suggests they are void functions.

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

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

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

    return 0;
}
```

in fact trying to return a value in void() will lead to compiler error

using function parameters to produce arguments

a **function parameter** is a variable used in the header of a function. An **argument** is a value that is passed *from* the caller *to* the function when a function call is made. 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**.

```cpp
#include <iostream>

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

    return input;
}

void printDouble(int value) // This function now has an integer parameter
{
    std::cout << value << " doubled is: " << value * 2 << '\n';
}

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

    printDouble(num);

    return 0;
}
```

we can also use **return values as function arguments** so for example there is a function called getValueFromUser() and printValue() so the statement

printValue(getValueFromUser()); //gets value from user and prints it

function return values can also be used as output statement

```cpp
    std::cout << add(1, multiply(2, 3)) << '\n'; // evaluates 1 + (2 * 3)
    std::cout << add(1, add(2, 3)) << '\n'; // evaluates 1 + (2 + 3)
```

local variables

variables created inside **function** or **function parameters** or a **block of code ({})** can only be used locally or inside the function/block and they go **"out of scope"** and are "destroyed" from memory(**duration of identifier ends**) once the function/block ends with curly braces whereas functions/variables declared outside main() or any function are known a **global variables**
local variables have **no linkage** which means that declaration in different scopes can only refer to the same object

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

    {
        int x { 3 }; // this declaration of x refers to a different object than the previous x
    }

    return 0;
}
```

x{3} will **nameshadow** x{2} in any call inside the inner code block.

This is why its good practice to declare variables only in the code blocks we use them to reduce the complexity of the program by reducing the number of active variables

global variables

identifiers declared in the global namespace have **global namespace scope** (commonly called **global scope**, and sometimes informally called **file scope**), which means they are visible from the point of declaration until the end of the *file* in which they are declared.

global variables are created when the program starts (before `main()` begins execution), and destroyed when it ends. This is called **static duration**. Variables with *static duration* are sometimes called **static variables**.In general developers name global variable identifiers with “g” or “g_” to indicate that they are global.

global variables without any initialisation are also zero initialised.
int g_x; outside any function will be initialised to **0** by default (unlike local variables that require list initialisation to be zero-initialised int l_x {};)

local variables can shadow global variables

```cpp
#include <iostream>
int value { 5 }; // global variable

void foo()
{
    std::cout << "global variable value: " << value << '\n'; // value is not shadowed here, so this refers to the global value
}

int main()
{
    int value { 7 }; // hides the global variable value until the end of this block

    ++value; // increments local value, not global value

    std::cout << "local variable value: " << value << '\n';

    foo();

    return 0;
} // local value is destroyed
```

This code prints:

```
local variable value: 8
global variable value: 5
```

https://www.learncpp.com/cpp-tutorial/function-pointers/
https://www.learncpp.com/cpp-tutorial/the-stack-and-the-heap/
https://www.learncpp.com/cpp-tutorial/recursion/
https://www.learncpp.com/cpp-tutorial/command-line-arguments/
https://www.learncpp.com/cpp-tutorial/ellipsis-and-why-to-avoid-them/
https://www.learncpp.com/cpp-tutorial/introduction-to-lambdas-anonymous-functions/
https://www.learncpp.com/cpp-tutorial/lambda-captures/

using functions effectively for DRY

typically, when learning C++, you will write a lot of programs that involve 3 subtasks:

1. Reading inputs from the user
2. Calculating a value from the inputs
3. Printing the calculated value
4.

for trivial programs (e.g. less than 20 lines of code), some or all of these can be done in function *main*. However, for longer programs (or just for practice) each of these is a good candidate for an individual function.

---

pre-processor

prior to compilation, each code (.cpp) file goes through a **pre-processing** phase. Its main function include stripping comments from the code and looking for pre-processor directives which are instructions that start with “#” which tell it perform text manipulation tasks. Pre-processors don’t understand C++ syntax at all.

#define

#define is used to create macros which changes input text into certain output replacements and traditionally the identifier is types in CAPITAL LETTERS separated by underscores. #define values will stay relevant throughout the code regardless of where they are defined as pre-processor can't understand C++ syntax.

```cpp
#include <iostream>

#define MY_NAME "Alex"

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

    return 0;
}
```

here <<MY_NAME<< is converted to << Alex << by pre-processor.

#define only defines identifiers till the end of the file and you need to include the file, to increase its scope of relevancy

Alex.h:

```cpp
#define MY_NAME "Alex"
```

main.cpp:

```cpp
#include "Alex.h" // copies #define MY_NAME from Alex.h here
#include <iostream>

int main()
{
    std::cout << "My name is: " << MY_NAME << '\n'; // preprocessor replaces MY_NAME with "Alex"

    return 0;
}
```

this form of object macro substitution was primarily used in C and is now **nothing but legacy code,** however object macro substitution without replacement text are generally considered acceptable.

#define MY_NAME

This replaces any further occurrence of this identifier with "nothing!"

#ifdef | #endif | #ifndef

The *conditional compilation* pre-processor directives allow you to specify under what conditions something will or won't compile.

```cpp
#include <iostream>

#define PRINT_JOE

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

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

    return 0;
}
```

Since PRINT_JOE has been defined, "Joe" will be printed

```cpp
#include <iostream>

int main()
{
#ifndef PRINT_BOB
    std::cout << "Bob\n";
#endif

    return 0;
}
```

#ifndef is opposite to #ifdef and therefore "Bob" will be printed

#If 0

#If0 is often used to comment out blocks of code we don't need to be compiled, especially for multi-line comments that cannot be nested between actual blocks of code that also require to be removed.

```cpp
#include <iostream>

int main()
{
    std::cout << "Joe\n";

#if 0 // Don't compile anything starting here
    std::cout << "Bob\n";
    /* Some
     * multi-line
     * comment here
     */
    std::cout << "Steve\n";
#endif // until this point

    return 0;
}
```

#if 1 instead of #if 0 will temporarily re-enable the code within its limits

#include

when we #include a file, the pre-processor replaces "#include directive" with contents of file we included

```cpp
#include <iostream>

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

Here we included the definition of std::cout with <iostream>, which would be important for this codefile to run as the compiler has no way of knowing what "std::cout" means. In absence of this mechanism, We would have to know and then define all the declarations related to std::cout for all the files that use the iostream.

Here's a brief overview of what `<iostream>` includes:

1. **Input/output stream objects**: `std::cin`, `std::cout`, `std::cerr`, `std::clog`.
2. **Stream manipulators**: These are used to modify the behavior of streams, such as `std::endl`, `std::setw`, `std::hex`, etc.
3. **Stream classes**: Classes like `std::istream`, `std::ostream`, `std::ifstream`, `std::ofstream`, `std::stringstream`, etc., which provide functionality for reading from and writing to different types of streams (e.g., files, strings).
4. **Input/output functions**: Functions like `std::getline`, `std::getline`, `std::put`, `std::get`, etc., which are used for input/output operations.

Additionally, `<iostream>` also includes other related standard headers such as `<ios>`, `<streambuf>`, `<istream>`, `<ostream>`, and `<iomanip>`, which provide more functionalities related to input/output operations and stream manipulations.

header files

All the files in the project are linked by a linker during the compilation process. This means that functions defined in different object files can be used in the main object file as long as

there is forward declaration at the beginning of the object file you want to utilise the function in. 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. We use a **header file** which includes all the forward declarations in one place convenient to use, whenever we need them. They usually have a **.h suffix** instead of .cpp by convention and #including .cpp files is non-conventional and could lead to naming collisions.

add.h

```
int add(int x, int y);
```

main.cpp

```
#include "add.h" // Contents of add.h copied here
#include <iostream>

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

    return 0;
}
```

add.cpp

```
#include "add.h" /* we include add.h in all files by convention, this is called paired header and is done to catch errors during compile time instead of link time. Some example of source files even recquire contents from paired header to run and hence it is important to include it*/

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

we only use header files to forward declare functions and not define functions in them

this is usually done to avoid multiple definitions of functions or variables as the linker will see the headerfile definition and definition in all cpp files having the header file which would violate ODR (One Definition Rule).

angle brackets(<>) vs double quotes (“”)

it’s possible to have header files with same names in different directories & that’s why we use them to provide clues to pre-processor on where to search for the file.

When we use angled brackets, we're telling the preprocessor that this is a header file we didn't write ourselves. The preprocessor will search for the header only in the directories specified by the `include directories`. The `include directories` are configured as part of your project/IDE settings/compiler settings, and typically default to the directories containing the header files that come with your compiler and/or OS. The preprocessor will not search for the header file in your project's source code directory.

When we use double-quotes, we're telling the preprocessor that this is a header file that we wrote. The preprocessor will first search for the header file in the current directory. If it can't find a matching header there, it will then search the `include directories`.

scope of #definition for #included files

function.cpp:

```
#include <iostream>

void doSomething()
{
#ifdef PRINT
    std::cout << "Printing!\n";
#endif
#ifndef PRINT
    std::cout << "Not printing!\n";
#endif
}
```

main.cpp:

```
void doSomething(); // forward declaration for function doSomething()

#define PRINT

int main()
{
    doSomething();

    return 0;
}
```

this will print Not printing! As #define PRINT only stays in relevance till the end of the file(ie main.cpp), therefore it was never included in the "scope of relevancy", as far as function.cpp is concerned.

header guards

since programs that define a function more than once will cause compile errors. 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:

```
int getSquareSides()
{
    return 4;
}
```

wave.h:

```
#include "square.h"
```

main.cpp:

```
#include "square.h"
#include "wave.h"

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

this innocent looking program won't run. The pre-processor will convert the main.cpp into –

```
int getSquareSides()  // from square.h
{
    return 4;
}

int getSquareSides() // from wave.h (via square.h)
{
    return 4;
}

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

which obviously will lead to a compile error due to int getsquaresides() getting defined twice
we can prevent this by using conditional compilation pre-processors called **header guards**.

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

wave.h:

```
#ifndef WAVE_H
#define WAVE_H

#include "square.h"

#endif
```

main.cpp:

```
#include "square.h"
#include "wave.h"

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

preprocessors this time, converts main.cpp into

```
// Square.h included from main.cpp
#ifndef SQUARE_H // square.h included from main.cpp
#define SQUARE_H // SQUARE_H gets defined here

// and all this content gets included
int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H

#ifndef WAVE_H // wave.h included from main.cpp
#define WAVE_H
#ifndef SQUARE_H // square.h included from wave.h, SQUARE_H is already defined from above
#define SQUARE_H // so none of this content gets included

int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H
#endif // WAVE_H

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

Header guards prevent duplicate inclusions because the first time a guard is encountered, the guard macro isn't defined, so the guarded content is included. Past that point, the guard macro is defined, so any subsequent copies of the guarded content are excluded.

#pragma once header guards

Modern compilers support a simpler, alternate form of header guards using the `#pragma` pre-processor directive in header files

```
1 #pragma once
2
3 // your code here
```

one problem with this is that double inclusion of header files can dupe the system as it will fail to recognise the same contents are repeated twice. This is fairly common with many header files having other header files #included inside them.

#pragma once is not defined by C++ standard and some compilers may not compile it. Google uses traditional header guards as well for the same reason. 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++

user-defined namespaces

A naming collision occurs when two identical identifiers are introduced into the same scope, and the compiler can't disambiguate which one to use. When this happens, compiler or linker will produce an error because they do not have enough information to resolve the ambiguity. We can obviously fix this problem by renaming one of the variable/function names but we would also need to change the name of all the function calls and variable uses in the code which can be a pain & subject to error.

C++ allows us to define our own namespaces via the `namespace` keyword. Namespaces that you create in your own programs are casually called **user-defined namespaces** (though it would be more accurate to call them **program-defined namespaces**).

foo.cpp:

```
1  namespace Foo // define a namespace named Foo
2  {
3      // This doSomething() belongs to namespace Foo
4      int doSomething(int x, int y)
5      {
6          return x + y;
7      }
8  }
```

goo.cpp:

```
1  namespace Goo // define a namespace named Goo
2  {
3      // This doSomething() belongs to namespace Goo
4      int doSomething(int x, int y)
5      {
6          return x - y;
7      }
8  }
```

now both doSomething() functions are not in global space and the linker won't be able to find it unless we use a **scoperesolution operator (::)** or **using statement**.

```
std::cout << Foo::doSomething(4, 3) << '\n';
return 0;
```

If a function call statement is present inside another function being called in main() that further belongs to a namespace. The function is initially found within the same namespace, then searched in other namespaces and finally searched for in the global namespace

If a identifier(variable/function name) is called with scope resolution operator with no namespace mentioned, the identifier is looked for in the global namespace. This is also a

good way to use global variables and overshadow, nameshadowing done by local variables.

```cpp
#include <iostream>

void print() // this print() lives in the global namespace
{
    std::cout << " there\n";
}

namespace Foo
{
    void print() // this print() lives in the Foo namespace
    {
        std::cout << "Hello";
    }

    void printHelloThere()
    {
        print();   // calls print() in Foo namespace
        ::print(); // calls print() in global namespace
    }
}

int main()
{
    Foo::printHelloThere();

    return 0;
}
```

This prints:

```
Hello there
```

The forward declaration in header files for namespace functions need to be written inside namespaces as well.

add.h

```cpp
#ifndef ADD_H
#define ADD_H

namespace BasicMath
{
    // function add() is part of namespace BasicMath
    int add(int x, int y);
}

#endif
```

add.cpp

```cpp
#include "add.h"

namespace BasicMath
{
    // define the function add() inside namespace BasicMath
    int add(int x, int y)
    {
        return x + y;
    }
}
```

main.cpp

```cpp
#include "add.h" // for BasicMath::add()

#include <iostream>

int main()
{
    std::cout << BasicMath::add(4, 3) << '\n';

    return 0;
}
```

It is **legal to have multiple namespace with same name across different files** and all contents will be united under the single common namespace.

nested namespaces are legal and have two methods of initialisation

```
#include <iostream>

namespace Foo
{
    namespace Goo // Goo is a namespace inside the Foo namespace
    {
        int add(int x, int y)
        {
            return x + y;
        }
    }
}

int main()
{
    std::cout << Foo::Goo::add(1, 2) << '\n';
    return 0;
}
```

```
#include <iostream>

namespace Foo::Goo // Goo is a namespace inside the Foo namespace (C++17 style)
{
    int add(int x, int y)
    {
        return x + y;
    }
}

int main()
{
    std::cout << Foo::Goo::add(1, 2) << '\n';
    return 0;
}
```

we can also create functions inside of Foo namespace separately along with Foo::Goo without any issue

```
namespace Foo
{
    void someFcn() {} // This function is in Foo only
}
```

since nested namespaces have complex names we can create **aliases** to call them without typing the whole ordeal.

```
int main()
{
    namespace Active = Foo::Goo; // active now refers to Foo::Goo

    std::cout << Active::add(1, 2) << '\n'; // This is really Foo::Goo::add()

    return 0;
} // The Active alias ends here
```

this is helpful in cases of a change, we are not forced change each instance of Foo::Goo but just initiate the alias to some other namespace

```
int main()
{
    namespace Active = V2; // active now refers to V2

    std::cout << Active::add(1, 2) << '\n'; // We don't have to change this

    return 0;
}
```

variables declared inside user defined namespace are also global variables and can be called within the global scope.

```
#include <iostream>

namespace Foo // Foo is defined in the global scope
{
    int g_x {}; // g_x is now inside the Foo namespace, but is still a global variable
}

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

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

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

    return 0;
}
```

unnamed (anonymous) namespaces

An **unnamed namespace** (also called an **anonymous namespace**) is a namespace that is defined without a name

```
#include <iostream>

namespace // unnamed namespace
{
    void doSomething() // can only be accessed in this file
    {
        std::cout << "v1\n";
    }
}

int main()
{
    doSomething(); // we can call doSomething() without a namespace prefix

    return 0;
}
```

identifiers of unnamed namespace act belong to the global namespace. This might make unnamed namespaces seem useless but the identifiers in a anonymous namespace are internally linked which is useful for creating **static functions**. Unnamed namespaces are typically used when a lot of content is required to be localised to a file without individually having to mark each declaration as static.

error

software errors are prevalent. It's easy to make them, and it's hard to find them. learning to find and remove bugs in the programs you write is an extremely important part of being a successful programmer. There are four type of errors including compile-time error, runtime error, syntax error, semantic error.

compile-Time Error encompasses issues that happen during the compilation process for example assigning wrong datatypes or using undefined variables/functions

runtime Error happens when the code is compiled but the generated result is not what we want to see from the code.

a **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. This includes errors such as missing semicolons, using undeclared variables, mismatched parentheses or braces, etc...

a **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended. Sometimes these will cause your program to crash, such as in the case of division by zero

debugging

you've written a program, and it's not working correctly -- the code compiles fine, but when you run it, you're getting an incorrect result. You must have a semantic error somewhere. How can you find it? If you've been following best practices by writing a little bit of code and then testing it, you may have a good idea where your error is. Or you may have no clue at all. All bugs stem from a simple premise: Something that you thought was correct, isn't. Actually figuring out where that error is can be challenging.

let's use a real-life analogy here. Let's say one evening, you go to get some ice from the ice dispenser in your freezer. You put your cup up to the dispenser, press the lever, and ... nothing comes out. Uh oh. You've discovered some kind of defect. What would you do? You'd probably start an investigation to see if you could identify the root cause of the issue.

find the root cause: Since you hear the ice dispenser trying to deliver ice, it's probably not the ice delivery mechanism itself. So you open the freezer, and examine the ice tray. No ice. Is that the root cause of the issue? No, it's another symptom. After further examination, you determine that the ice maker does not appear to be making ice. Is the problem the ice maker or something else? The freezer is still cold, the water line isn't clogged, and everything else seems to be working, so you conclude that the root cause is that the ice maker is non-functional.

understand the problem: This is simple in this case. A broken ice maker won't make ice.
determine a fix: At this point, you have several options for a fix: You could work around the issue (buy bags of ice from the store). You could try to diagnose the ice-maker further, to see if there's a part that can be repaired. You could buy a new ice maker and install it in place of the current one. Or you could buy a new freezer. You decide to buy a new ice maker.
repair the issue: Once the ice maker has arrived, you install it.
retest: After turning the electricity back on and waiting overnight, your new ice maker starts making ice. No new issues are discovered.

as programs get more complex, finding issues by code inspection becomes more complex as well. First, there's a lot more code to look at. Looking at every line of code in a program that is thousands of lines long can take a really long time (not to mention it's incredibly boring). Second, the code itself tends to be more complex, with more possible places for things to go wrong. Third, the code's behaviour may not give you many clues as to where things are going wrong. If you wrote a program to output stock recommendations and it actually output nothing at all, you probably wouldn't have much of a lead on where to start looking for the problem.

finally, bugs can be caused by making bad assumptions. It's almost impossible to visually spot a bug caused by a bad assumption, because you're likely to make the same bad assumption when inspecting the code, and not notice the error.

reproducing the problem

the first and most important step in finding a problem is to be able to *reproduce the problem*. Reproducing the problem means making the problem appear in a consistent manner. The reason is simple: it's extremely hard to find an issue unless you can observe it occurring. Back to our ice dispenser analogy -- let's say one day your friend tells you that your ice dispenser isn't working. You go to look at it, and it works fine. How would you diagnose the problem? It would be very difficult. However, if you could actually see the issue of the ice dispenser not working, then you could start to diagnose why it wasn't working much more effectively.

if a software issue is blatant (e.g. the program crashes in the same place every time you run it) then reproducing the problem can be trivial. However, sometimes reproducing an issue can be a lot more difficult. The problem may only occur on certain computers, or in particular circumstances (e.g. when the user enters certain input). In such cases, generating a set of reproduction steps can be helpful. **Reproduction steps** are a list of clear and precise steps that can be followed to cause an issue to recur with a high level of predictability. The goal is to be able to cause the issue to reoccur as much as possible, so we can run our program over and over and look for clues to determine what's causing the problem. If the issue can be reproduced 100% of the time, that's ideal, but less than 100% reproducibility can be okay. An issue that occurs only 50% of the time simply means it'll take twice as long to diagnose the issue, as half the time the program won't exhibit the problem and thus not contribute any useful diagnostic information.

homing on the issues

once we can reasonably reproduce the problem, the next step is to figure out where in the code the problem is. In the worst case, we may have no idea where the bug is. However, we do know that the problem must be somewhere in the code that executes between the beginning of the program and the point where the program exhibits the first incorrect symptom that we can observe. That at least rules out the parts of the program that execute after the first observable symptom. But that still potentially leaves a lot of code to cover. To diagnose the issue, we'll make some educated guesses about where the problem is, with the goal of homing in on the problem quickly.

often, whatever it was that caused us to notice the problem will give us an initial guess that's close to where the actual problem is.

debugging tactic #1: commenting out your code

let's start with an easy one. If your program is exhibiting erroneous behavior, one way to reduce the amount of code you have to search through is to comment some code out and see if the issue persists. If the issue remains unchanged, the commented out code probably wasn't responsible.

debugging tactic #2 : validating your code flow

another problem common in more complex programs is that the program is calling a function too many or too few times (including not at all).

in such cases, it can be helpful to place statements at the top of your functions to print the function's name. That way, when the program runs, you can see which functions are getting called.

debugging tactic #3 : printing code values

with some types of bugs, the program may be calculating or passing the wrong value.We can also output the value of variables (including parameters) or expressions to ensure that they are correct.

since debug statements can clutter your codebase & output of the program, introduce new bugs and have to be removed after you're done with them making them un-reusable. One way to make it easier to disable and enable debugging throughout your program is to make your debugging statements conditional using pre-processor directives.

```
#include <iostream>

#define ENABLE_DEBUG // comment out to disable debugging

int getUserInput()
{
#ifdef ENABLE_DEBUG
std::cerr << "getUserInput() called\n";
#endif
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;
    return x;
}

int main()
{
#ifdef ENABLE_DEBUG
std::cerr << "main() called\n";
#endif
    int x{ getUserInput() };
    std::cout << "You entered: " << x << '\n';

    return 0;
}
```

now we can enable debugging simply by commenting / uncommenting *#define ENABLE_DEBUG*. this allows us to reuse previously added debug statements and then just disable them when we're done with them, rather than having to actually remove them from the code. If this were a

multi-file program, the #define ENABLE_DEBUG would go in a header file that's included into all code files so we can comment / uncomment the #define in a single location and have it propagate to all code files.

---

words of advice when writing programs

keep your programs simple to start. Often new programmers have a grand vision for all the things they want their program to do. "I want to write a role-playing game with graphics and sound and random monsters and dungeons, with a town you can visit to sell the items that you find in the dungeon". If you try to write something too complex to start, you will become overwhelmed and discouraged at your lack of progress. Instead, make your first goal as simple as possible, something that is definitely within your reach. For example, "I want to be able to display a 2-dimensional field on the screen".

add features over time. Once you have your simple program working and working well, then you can add features to it. For example, once you can display your field, add a character who can walk around. Once you can walk around, add walls that can impede your progress. Once you have walls, build a simple town out of them. Once you have a town, add merchants. By adding each feature incrementally your program will get progressively more complex without overwhelming you in the process.

focus on one area at a time. Don't try to code everything at once, and don't divide your attention across multiple tasks. Focus on one task at a time. It is much better to have one working task and five that haven't been started yet than six partially-working tasks. If you split your attention, you are more likely to make mistakes and forget important details.

test each piece of code as you go. New programmers will often write the entire program in one pass. Then when they compile it for the first time, the compiler reports hundreds of errors. This can not only be intimidating, if your code doesn't work, it may be hard to figure out why. Instead, write a piece of code, and then compile and test it immediately. If it doesn't work, you'll know exactly where the problem is, and it will be easy to fix. Once you are sure that the code works, move to the next piece and repeat. It may take longer to finish writing your code, but when you are done the whole thing
should work, and you won't have to spend twice as long trying to figure out why it doesn't.

don't invest in perfecting early code. The first draft of a feature (or program) is rarely good. Furthermore, programs tend to evolve over time, as you add capabilities and find better ways to structure things. If you invest too early in polishing your code (adding lots of documentation, full compliance with best practices, making optimizations), you risk losing all of that investment when a code change is necessary. Instead, get your features minimally working and then move on. As you gain confidence in your solutions, apply successive layers of polish. Don't aim for perfect -- non-trivial programs are never perfect, and there's always something more that could be done to improve them. Get to "good enough" and move on.

optimize for maintainability, not performance. There is a famous quote (by Donald Knuth) that says "premature optimization is the root of all evil". New programmers often spend far too much time thinking about how to micro-optimize their code (e.g. trying to figure out which of 2 statements is faster). This rarely matters. Most performance benefits come from good program structure, using the right tools and capabilities for the problem at hand, and following best practices. Additional time should be used to improve the maintainability of your code. Find

redundancy and remove it. Split up long functions into shorter ones. Replace awkward or hard to use code with something better. The end result will be code that is easier to improve and optimize later (after you've determined where optimization is actually needed) and fewer bugs.