UPCISS

# C++

Complete Tutorial with Free PDF Notes and video lecture.

**Video Tutorial**

**C++** Full Video Playlist Available on YouTube Channel **UPCISS**

**Free** Online Computer Classes on YouTube Channel **UPCISS**

**www.youtube.com/upciss**

**For free PDF Notes**

**Our Website: www.upcissyoutube.com**

## Index

# C++ Introduction

## What is C++?

C++ is a cross-platform language that can be used to create high-performance applications.

C++ was developed by Bjarne Stroustrup, as an extension to the C language.

C++ gives programmers a high level of control over system resources and memory.

The language was updated 4 major times in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20.

## Why Use C++

C++ is one of the world's most popular programming languages.

C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

C++ is fun and easy to learn!

As C++ is close to C, C# and Java, it makes it easy for programmers to switch to C++ or vice versa.

## Difference between C and C++

C++ was developed as an extension of C, and both languages have almost the same syntax.

The main difference between C and C++ is that C++ support classes and objects, while C does not.

## Get Started

This tutorial will teach you the basics of C++.

It is not necessary to have any prior programming experience.

To start using C++, you need two things:

- A text editor, like Notepad, to write C++ code
- A compiler, like GCC, to translate the C++ code into a language that the computer will understand

There are many text editors and compilers to choose from. In this tutorial, we will use an IDE (see below).

## C++ Install IDE

An IDE (Integrated Development Environment) is used to edit AND compile the code.

Popular IDE's include Code::Blocks, Eclipse, and Visual Studio. These are all free, and they can be used to both edit and debug C++ code.

**Note:** Web-based IDE's can work as well, but functionality is limited.

We will use **Code::Blocks** in our tutorial, which we believe is a good place to start.

You can find the latest version of Codeblocks at [http://www.codeblocks.org/](http://www.codeblocks.org/). Download the `mingw-setup.exe` file, which will install the text editor with a compiler.

## C++ Quickstart

Let's create our first C++ file.

Open Codeblocks and go to **File > New > Empty File**.

Write the following C++ code and save the file as `myfirstprogram.cpp` (**File > Save File as**):

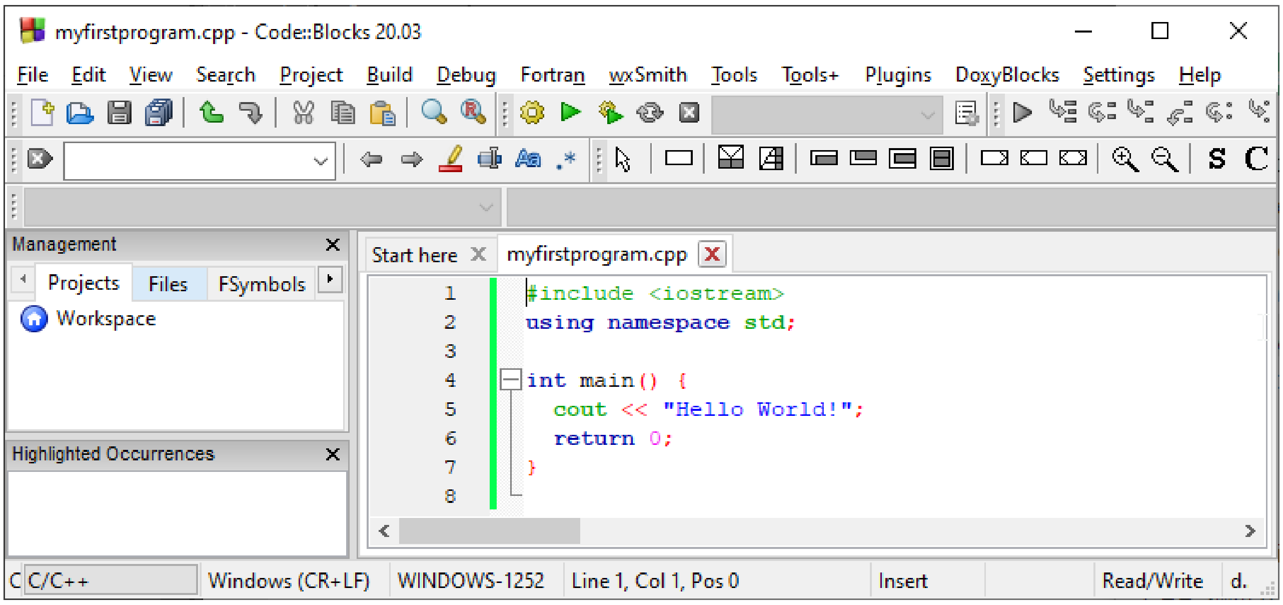
***myfirstprogram.cpp***

```cpp
#include <iostream>
using namespace std;

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

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on how to run the code.

In Codeblocks, it should look like this:


myfirstprogram.cpp - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Files FSymbols
Workspace
Highlighted Occurrences
Start here
myfirstprogram.cpp
C/C++ Windows (CR+LF) WINDOWS-1252 Line 1, Col 1, Pos 0 Insert Read/Write

```
#include <iostream>
using namespace std;

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

```

Then, go to **Build > Build and Run** to run (execute) the program. The result will look something to this:

```
Hello World!
Process returned 0 (0x0) execution time : 0.011 s
Press any key to continue.
```

**Congratulations**! You have now written and executed your first C++ program.

# C++ Syntax

## C++ Syntax

Let's break up the following code to understand it better:

### Example

```
#include <iostream>
using namespace std;

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

### Example explained

**Line 1:** `#include <iostream>` is a **header file library** that lets us work with input and output objects, such as `cout` (used in line 5). Header files add functionality to C++ programs.

**Line 2:** `using namespace std` means that we can use names for objects and variables from the standard library.

Don't worry if you don't understand how `#include <iostream>` and `using namespace std` works. Just think of it as something that (almost) always appears in your program.

**Line 3:** A blank line. C++ ignores white space. But we use it to make the code more readable.

**Line 4:** Another thing that always appear in a C++ program, is `int main()`. This is called a **function**. Any code inside its curly brackets `{}` will be executed.

**Line 5:** `cout` (pronounced "see-out") is an **object** used together with the *insertion operator* (`<<`) to output/print text. In our example it will output "Hello World!".

**Note:** Every C++ statement ends with a semicolon `;`.

**Note:** The body of `int main()` could also been written as:
`int main () { cout << "Hello World! "; return 0; }`

**Remember:** The compiler ignores white spaces. However, multiple lines makes the code more readable.

**Line 6:** `return 0` ends the main function.

**Line 7:** Do not forget to add the closing curly bracket `}` to actually end the main function.

## Omitting Namespace

You might see some C++ programs that runs without the standard namespace library. The `using namespace std` line can be omitted and replaced with the `std` keyword, followed by the `::` operator for some objects:

### Example

```
#include <iostream>

int main() {
  std::cout << "Hello World!";
  return 0;
}
```

It is up to you if you want to include the standard namespace library or not.

## C++ Output (Print Text)

The `cout` object, together with the `<<` operator, is used to output values/print text:

### Example

```
#include <iostream>
using namespace std;
```

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

You can add as many `cout` objects as you want. However, note that it does not insert a new line at the end of the output:

## Example

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!";
  cout << "I am learning C++";
  return 0;
}
```

## New Lines

To insert a new line, you can use the `\n` character:

## Example

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World! \n";
  cout << "I am learning C++";
  return 0;
}
```

**Tip:** Two `\n` characters after each other will create a blank line:

## Example

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World! \n\n";
  cout << "I am learning C++";
  return 0;
}
```

Another way to insert a new line, is with the `endl` manipulator:

## Example

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  cout << "I am learning C++";
  return 0;
}
```

Both \n and endl are used to break lines. However, \n is most used.

### ***But what is \n exactly?***

The newline character (\n) is called an **escape sequence**, and it forces the cursor to change its position to the beginning of the next line on the screen. This results in a new line.

Examples of other valid escape sequences are:

| Escape Sequence | Description |
|---|---|
| \t | Creates a horizontal tab |
| \\ | Inserts a backslash character (\) |
| \" | Inserts a double quote character |

## C++ Comments

Comments can be used to explain C++ code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Comments can be singled-lined or multi-lined.

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by the compiler (will not be executed).

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by the compiler:

## C++ Variables

Variables are containers for storing data values.

In C++, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

## Declaring (Creating) Variables

To create a variable, specify the type and assign it a value:

### Syntax

```
type variableName = value;
```

Where *type* is one of C++ types (such as `int`), and *variableName* is the name of the variable (such as **x** or **myName**). The **equal sign** is used to assign values to the variable.

To create a variable that should store a number, look at the following example:

### Example

Create a variable called **myNum** of type `int` and assign it the value **15**:

```
int myNum = 15;
cout << myNum;
```

You can also declare a variable without assigning the value, and assign the value later:

### Example

```
int myNum;
myNum = 15;
cout << myNum;
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

### Example

```
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
cout << myNum;  // Outputs 10
```

## Other Types

A demonstration of other data types:

**Example**

```
int myNum = 5;               // Integer (whole number without decimals)
double myFloatNum = 5.99;    // Floating point number (with decimals)
char myLetter = 'D';         // Character
string myText = "Hello";     // String (text)
bool myBoolean = true;       // Boolean (true or false)
```

## Display Variables

The `cout` object is used together with the `<<` operator to display variables.

To combine both text and a variable, separate them with the `<<` operator:

**Example**

```
int myAge = 35;
cout << "I am " << myAge << " years old.";
```

## Declare Many Variables

To declare more than one variable of the **same type**, use a comma-separated list:

**Example**

```
int x = 5, y = 6, z = 50;
cout << x + y + z;
```

## One Value to Multiple Variables

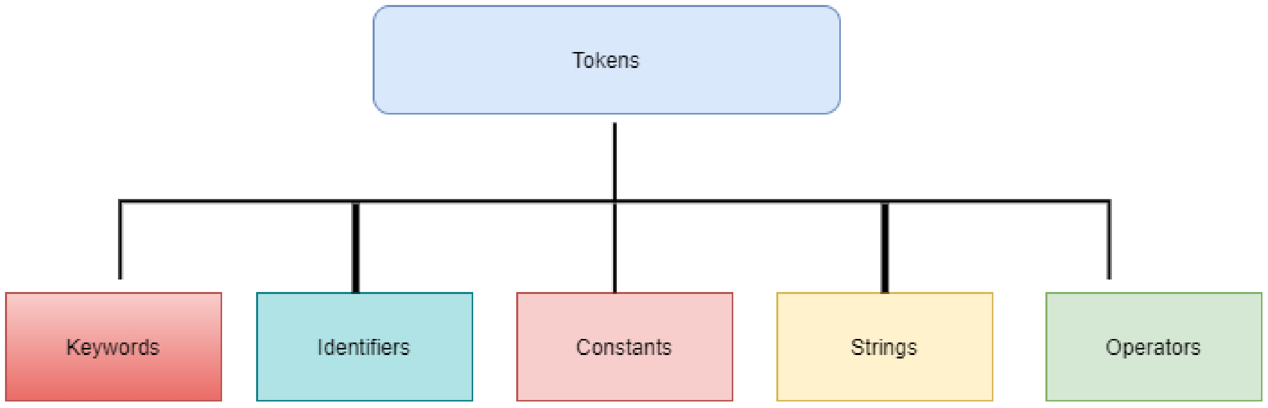
You can also assign the **same value** to multiple variables in one line:

**Example**

```
int x, y, z;
x = y = z = 50;
cout << x + y + z;
```

## Token

When the compiler is processing the source code of a C++ program, each group of characters separated by white space is called a token. Tokens are the smallest individual units in a program. A C++ program is written using tokens. It has the following tokens:



## Keywords

Keywords (also known as reserved words) have special meanings to the C++ compiler and are always written or typed in short (lower) cases. Keywords are words that the language uses for a special purpose, such as void, int, public, etc. It can't be used for a variable name or function name or any other identifiers. The total count of reserved keywords is 95. Below is the table for some commonly used C++ keywords.

| C++ Keyword | | | |
|---|---|---|---|
| asm | double | new | switch |
| auto | else | operator | template |
| break | enum | private | this |
| case | extern | protected | throw |
| catch | float | public | try |
| char | for | register | typedef |
| class | friend | return | union |
| const | goto | short | unsigned |
| continue | if | signed | virtual |
| default | inline | sizeof | void |
| delete | int | static | volatile |
| do | long | struct | while |

## C++ Identifiers

All C++ **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

**Note:** It is recommended to use descriptive names in order to create understandable and maintainable code:

## Example

```
// Good
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

The general rules for naming variables are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case-sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (like C++ keywords, such as int) cannot be used as names

## Constants

When you do not want others (or yourself) to change existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

## Example

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

You should always declare the variable as constant when you have values that are unlikely to change:

## Example

```
const int minutesPerHour = 60;
const float PI = 3.14;
```

When you declare a constant variable, it must be assigned with a value:

## Example

Like this:

```
const int minutesPerHour = 60;
```

This however, **will not work**:

```
const int minutesPerHour;
minutesPerHour = 60; // error
```

The constant variables in c are immutable after its definition, i.e., they can be initialized only once in the whole program. After that, we cannot modify the value stored inside that variable.

## C++ User Input

You have already learned that `cout` is used to output (print) values. Now we will use `cin` to get user input.

`cin` is a predefined variable that reads data from the keyboard with the extraction operator (`>>`).

In the following example, the user can input a number, which is stored in the variable `x`. Then we print the value of `x`:

**Example**

```
int x;
cout << "Type a number: "; // Type a number and press enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the input value
```

***Good To Know***

`cout` is pronounced "see-out". Used for **output**, and uses the insertion operator (`<<`)

`cin` is pronounced "see-in". Used for **input**, and uses the extraction operator (`>>`)

## Creating a Simple Calculator

In this example, the user must input two numbers. Then we print the sum by calculating (adding) the two numbers:

**Example**

```
int x, y;
int sum;
cout << "Type a number: ";
cin >> x;
cout << "Type another number: ";
cin >> y;
sum = x + y;
cout << "Sum is: " << sum;
```

## C++ Data Types

As explained in the Variables chapter, a variable in C++ must be a specified data type:

**Example**

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99;     // Floating point number
```

```
double myDoubleNum = 9.98;    // Floating point number
char myLetter = 'D';          // Character
bool myBoolean = true;        // Boolean
string myText = "Hello";      // String
```

## Basic Data Types

The data type specifies the size and type of information the variable will store:

| Data Type | Size | Description |
|---|---|---|
| `boolean` | 1 byte | Stores true or false values |
| `char` | 1 byte | Stores a single character/letter/number, or ASCII values |
| `int` | 2 or 4 bytes | Stores whole numbers, without decimals |
| `float` | 4 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits |
| `double` | 8 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits |

## Numeric Types

Use `int` when you need to store a whole number without decimals, like 35 or 1000, and `float` or `double` when you need a floating point number (with decimals), like 9.99 or 3.14515.

### int

```
int myNum = 1000;
cout << myNum;
```

### float

```
float myNum = 5.75;
cout << myNum;
```

### double

```
double myNum = 19.99;
cout << myNum;
```

`float` vs. `double`

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `float` is only six or seven decimal digits, while `double` variables have a precision of about 15 digits. Therefore it is safer to use `double` for most calculations.

## Boolean Types

A boolean data type is declared with the `bool` keyword and can only take the values `true` or `false`.

When the value is returned, `true` = `1` and `false` = `0`.

### Example

```
bool isCodingFun = true;
bool isFishTasty = false;
cout << isCodingFun;  // Outputs 1 (true)
cout << isFishTasty;  // Outputs 0 (false)
```

## Character Types

The `char` data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

### Example

```
char myGrade = 'B';
cout << myGrade;
```

Alternatively, if you are familiar with ASCII, you can use ASCII values to display certain characters:

### Example

```
char a = 65, b = 66, c = 67;
cout << a;
cout << b;
cout << c;
```

## String Types

The `string` type is used to store a sequence of characters (text). This is not a built-in type, but it behaves like one in its most basic usage. String values must be surrounded by double quotes:

### Example

```
string greeting = "Hello";
cout << greeting;
```

To use strings, you must include an additional header file in the source code, the `<string>` library:

## Example

```
// Include the string library
#include <string>

// Create a string variable
string greeting = "Hello";

// Output string value
cout << greeting;
```

## C++ Operators

An **operator** is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality.

An operator operates the **operands**.

In the example below, we use the + **operator** to add together two values:

## Example

```
int x = 100 + 50;
```

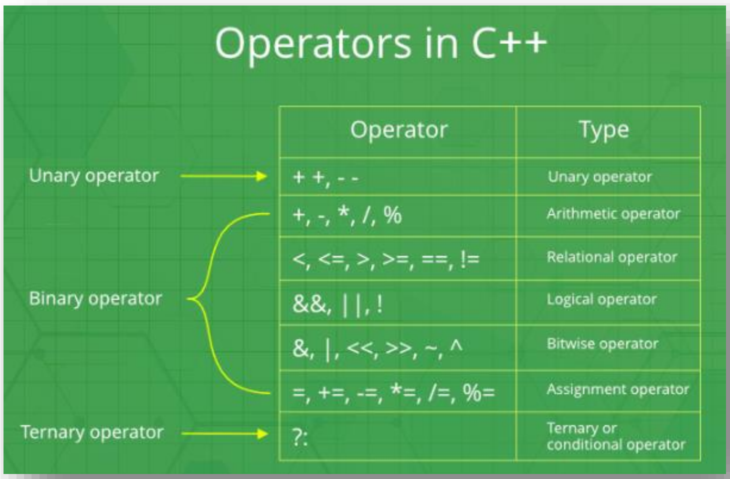
Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

## Example

```
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

C++ divides the operators into the following groups:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Ternary or Conditional Operators



Operators in C++

| | Operator | Type |
|---|---|---|
| Unary operator | ++, -- | Unary operator |
| Binary operator | +, -, *, /, % | Arithmetic operator |
| | <, <=, >, >=, ==, != | Relational operator |
| | &&, \|\|, ! | Logical operator |
| | &, \|, <<, >>, ~, ^ | Bitwise operator |
| | =, +=, -=, *=, /=, %= | Assignment operator |
| Ternary operator | ?: | Ternary or conditional operator |

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds together two values | x + y |
| - | Subtraction | Subtracts one value from another | x - y |
| * | Multiplication | Multiplies two values | x * y |
| / | Division | Divides one value by another | x / y |
| % | Modulus | Returns the division remainder | x % y |
| ++ | Increment | Increases the value of a variable by 1 | ++x |
| -- | Decrement | Decreases the value of a variable by 1 | --x |

## Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (`=`) to assign the value **10** to a variable called **x**:

### Example

```
int x = 10;
```

The **addition assignment** operator (`+=`) adds a value to a variable:

### Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |

|  |  |  |
|---|---|---|
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

## Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either `1` or `0`, which means **true** (1) or **false** (0). These values are known as **Boolean values**, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the **greater than** operator (`>`) to find out if 5 is greater than 3:

### Example

```
int x = 5;
int y = 3;
cout << (x > y); // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

| Operator | Name | Example |
|---|---|---|
| == | Equal to | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

## Logical Operators

As with comparison operators, you can also test for **true** (`1`) or **false** (`0`) values with **logical operators**.

Logical operators are used to determine the logic between variables or values:

| Operator | Name | Description | Example |
|---|---|---|---|

|  |  |  |  |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |

## Bitwise Operators

Bitwise Operators are the operators that are used to perform operations on the bit level on the integers. While performing this operation integers are considered as sequences of binary digits. In C++, we have various types of Bitwise Operators.

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise XOR (^)
4. Bitwise NOT (~)
5. Left Shift (<<)
6. Right Shift (>>)

```cpp
// C++ Program to demonstrate
// Bitwise Operator
#include <iostream>

using namespace std;

// Main function
int main()
{
    int a = 5; //  101
    int b = 3; //  011

    // Bitwise AND
    int bitwise_and = a & b;

    // Bitwise OR
    int bitwise_or = a | b;

    // Bitwise XOR
    int bitwise_xor = a ^ b;

    // Bitwise NOT
    int bitwise_not = ~a;

    // Bitwise Left Shift
    int left_shift = a << 2;

    // Bitwise Right Shift
    int right_shift = a >> 1;

      // Printing the Results of
    // Bitwise Operators
    cout << "AND: " << bitwise_and << endl;
    cout << "OR: " << bitwise_or << endl;
    cout << "XOR: " << bitwise_xor << endl;
    cout << "NOT a: " << bitwise_not << endl;
    cout << "Left Shift: " << left_shift << endl;
    cout << "Right Shift: " << right_shift << endl;

    return 0;
}
```

**Output:**

```
AND: 1
OR: 7
XOR: 6
NOT a: -6
Left Shift: 20
Right Shift: 2
```

## String Concatenation

The `+` operator can be used between strings to add them together to make a new string. This is called **concatenation**:

### Example

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
cout << fullName;
```

In the example above, we added a space after firstName to create a space between John and Doe on output. However, you could also add a space with quotes (`" "` or `' '`):

### Example

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
cout << fullName;
```

## Append

A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the `append()` function:

### Example

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
```

## Adding Numbers and Strings

WARNING!

C++ uses the `+` operator for both **addition** and **concatenation**.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

**Example**

```
int x = 10;
int y = 20;
int z = x + y;      // z will be 30 (an integer)
```

If you add two strings, the result will be a string concatenation:

**Example**

```
string x = "10";
string y = "20";
string z = x + y;   // z will be 1020 (a string)
```

If you try to add a number to a string, an error occurs:

**Example**

```
string x = "10";
int y = 20;
string z = x + y;
```

## String Length

To get the length of a string, use the `length()` function:

**Example**

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << "The length of the txt string is: " << txt.length();
```

**Tip:** You might see some C++ programs that use the `size()` function to get the length of a string. This is just an alias of `length()`. It is completely up to you if you want to use `length()` or `size()`:

**Example**

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << "The length of the txt string is: " << txt.size();
```

## Access Strings

You can access the characters in a string by referring to its index number inside square brackets `[]`.

This example prints the **first character** in **myString**:

### Example

```
string myString = "Hello";
cout << myString[0];
// Outputs H
```

**Note:** String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the **second character** in **myString**:

### Example

```
string myString = "Hello";
cout << myString[1];
// Outputs e
```

## Change String Characters

To change the value of a specific character in a string, refer to the index number, and use single quotes:

### Example

```
string myString = "Hello";
myString[0] = 'J';
cout << myString;
// Outputs Jello instead of Hello
```

## Strings - Special Characters

Because strings must be written within quotes, C++ will misunderstand this string, and generate an error:

```
string txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

| Escape character | Result | Description |
|---|---|---|
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

The sequence \" inserts a double quote in a string:

### Example

```
string txt = "We are the so-called \"Vikings\" from the north.";
```

The sequence \' inserts a single quote in a string:

### Example

```
string txt = "It\'s alright.";
```

The sequence \\ inserts a single backslash in a string:

### Example

```
string txt = "The character \\ is called backslash.";
```

## User Input Strings

It is possible to use the extraction operator `>>` on `cin` to store a string entered by a user:

### Example

```
string firstName;
cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard
cout << "Your name is: " << firstName;

// Type your first name: John
// Your name is: John
```

However, `cin` considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only store a single word (even if you type many words):

### Example

```
string fullName;
cout << "Type your full name: ";
cin >> fullName;
cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John
```

From the example above, you would expect the program to print "John Doe", but it only prints "John".

That's why, when working with strings, we often use the `getline()` function to read a line of text. It takes `cin` as the first parameter, and the string variable as second:

**Example**

```
string fullName;
cout << "Type your full name: ";
getline (cin, fullName);
cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John Doe
```

## C++ Math

C++ has many functions that allows you to perform mathematical tasks on numbers.

## Max and min

The `max(x,y)` function can be used to find the highest value of *x* and *y*:

**Example**

```
cout << max(5, 10);
```

And the `min(x,y)` function can be used to find the lowest value of *x* and *y*:

**Example**

```
cout << min(5, 10);
```

## C++ <cmath> Header

Other functions, such as `sqrt` (square root), `round` (rounds a number) and `log` (natural logarithm), can be found in the `<cmath>` header file:

**Example**

```
// Include the cmath library
#include <cmath>

cout << sqrt(64);
cout << round(2.6);
cout << log(2);
```

## C++ Math Functions

The `<cmath>` library has many functions that allow you to perform mathematical tasks on numbers.

A list of all math functions can be found in the table below:

| **Function** | **Description** |
|---|---|
| abs(x) | Returns the absolute value of x |
| acos(x) | Returns the arccosine of x, in radians |
| acosh(x) | Returns the hyperbolic arccosine of x |
| asin(x) | Returns the arcsine of x, in radians |
| asinh(x) | Returns the hyperbolic arcsine of x |
| atan(x) | Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians |
| atan2(y, x) | Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta) |
| atanh(x) | Returns the hyperbolic arctangent of x |
| cbrt(x) | Returns the cube root of x |
| ceil(x) | Returns the value of x rounded up to its nearest integer |
| copysign(x, y) | Returns the first floating point x with the sign of the second floating point y |
| cos(x) | Returns the cosine of x (x is in radians) |
| cosh(x) | Returns the hyperbolic cosine of x |
| exp(x) | Returns the value of $E^x$ |
| exp2(x) | Returns the value of $2^x$ |
| expm1(x) | Returns $e^x$-1 |
| erf(x) | Returns the value of the error function at x |
| erfc(x) | Returns the value of the complementary error function at x |
| fabs(x) | Returns the absolute value of a floating x |
| fdim(x) | Returns the positive difference between x and y |
| floor(x) | Returns the value of x rounded down to its nearest integer |
| fma(x, y, z) | Returns x*y+z without losing precision |
| fmax(x, y) | Returns the highest value of a floating x and y |

| | |
|---|---|
| fmin(x, y) | Returns the lowest value of a floating x and y |
| fmod(x, y) | Returns the floating point remainder of x/y |
| frexp(x, y) | With x expressed as $m*2^n$, returns the value of *m* (a value between 0.5 and 1.0) and writes the value of *n* to the memory at the pointer y |
| hypot(x, y) | Returns sqrt($x^2$ +$y^2$) without intermediate overflow or underflow |
| ilogb(x) | Returns the integer part of the floating-point base logarithm of x |
| ldexp(x, y) | Returns $x*2^y$ |
| lgamma(x) | Returns the logarithm of the absolute value of the gamma function at x |
| llrint(x) | Rounds x to a nearby integer and returns the result as a long long integer |
| llround(x) | Rounds x to the nearest integer and returns the result as a long long integer |
| log(x) | Returns the natural logarithm of x |
| log10(x) | Returns the base 10 logarithm of x |
| log1p(x) | Returns the natural logarithm of x+1 |
| log2(x) | Returns the base 2 logarithm of the absolute value of x |
| logb(x) | Returns the floating-point base logarithm of the absolute value of x |
| lrint(x) | Rounds x to a nearby integer and returns the result as a long integer |
| lround(x) | Rounds x to the nearest integer and returns the result as a long integer |
| modf(x, y) | Returns the decimal part of x and writes the integer part to the memory at the pointer y |
| nan(s) | Returns a NaN (Not a Number) value |
| nearbyint(x) | Returns x rounded to a nearby integer |
| nextafter(x, y) | Returns the closest floating point number to x in the direction of y |
| nexttoward(x, y) | Returns the closest floating point number to x in the direction of y |
| pow(x, y) | Returns the value of x to the power of y |

| | |
|---|---|
| remainder(x, y) | Return the remainder of x/y rounded to the nearest integer |
| remquo(x, y, z) | Calculates x/y rounded to the nearest integer, writes the result to the memory at the pointer z and returns the remainder. |
| rint(x) | Returns x rounded to a nearby integer |
| round(x) | Returns x rounded to the nearest integer |
| scalbln(x, y) | Returns x*$R^y$ (R is usually 2) |
| scalbn(x, y) | Returns x*$R^y$ (R is usually 2) |
| sin(x) | Returns the sine of x (x is in radians) |
| sinh(x) | Returns the hyperbolic sine of x |
| sqrt(x) | Returns the square root of x |
| tan(x) | Returns the tangent of x (x is in radians) |
| tanh(x) | Returns the hyperbolic tangent of x |
| tgamma(x) | Returns the value of the gamma function at x |
| trunc(x) | Returns the integer part of x |

# C++ Conditions and If Statements

You already know that C++ supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

C++ has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

## The if Statement

Use the `if` statement to specify a block of C++ code to be executed if a condition is `true`.

### Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `true`, print some text:

### Example

```
if (20 > 18) {
  cout << "20 is greater than 18";
}
```

We can also test variables:

### Example

```
int x = 20;
int y = 18;
if (x > y) {
  cout << "x is greater than y";
}
```

## The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `false`.

### Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

### Example

```
int time = 20;
if (time < 18) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
// Outputs "Good evening."
```

## The else if Statement

Use the `else if` statement to specify a new condition if the first condition is `false`.

### Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2
is true
} else {
  // block of code to be executed if the condition1 is false and condition2
is false
}
```

### Example

```
int time = 22;
if (time < 10) {
  cout << "Good morning.";
} else if (time < 20) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
// Outputs "Good evening."
```

## Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

### Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:

### Example

```
int time = 20;
if (time < 18) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
```

You can simply write:

## Example

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result;
```

# C++ Switch Statements

Use the `switch` statement to select one of many code blocks to be executed.

## Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `break` and `default` keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

## Example

```
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}
// Outputs "Thursday" (day 4)
```

## The break Keyword

When C++ reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

ADVERTISEMENT

## The default Keyword

The `default` keyword specifies some code to run if there is no case match:

### Example

```
int day = 4;
switch (day) {
  case 6:
    cout << "Today is Saturday";
    break;
  case 7:
    cout << "Today is Sunday";
    break;
  default:
    cout << "Looking forward to the Weekend";
}
// Outputs "Looking forward to the Weekend"
```

## C++ Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

## C++ While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

### Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (`i`) is less than 5:

### Example

```
int i = 0;
while (i < 5) {
  cout << i << "\n";
  i++;
}
```

**Note:** Do not forget to increase the variable used in the condition, otherwise the loop will never end!

## The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

### Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

### Example

```
int i = 0;
do {
  cout << i << "\n";
  i++;
}
while (i < 5);
```

## C++ For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

### Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

### Example

```
for (int i = 0; i < 5; i++) {
  cout << i << "\n";
}
```

#### *Example explained*

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

## Another Example

This example will only print even values between 0 and 10:

## Example

```
for (int i = 0; i <= 10; i = i + 2) {
  cout << i << "\n";
}
```

## Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

## Example

```
// Outer loop
for (int i = 1; i <= 2; ++i) {
  cout << "Outer: " << i << "\n"; // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; ++j) {
    cout << " Inner: " << j << "\n"; // Executes 6 times (2 * 3)
  }
}
```

## The foreach Loop

There is also a "**for-each** loop" (introduced in C++ version 11 (2011), which is used exclusively to loop through elements in an array (or other data sets):

## Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

The following example outputs all elements in an array, using a "**for-each** loop":

## Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

**Note:** Don't worry if you don't understand the example above. You will learn more about arrays in the C++ Arrays chapter.

## C++ Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when `i` is equal to 4:

### Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  cout << i << "\n";
}
```

## C++ Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

### Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  cout << i << "\n";
}
```

## C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of three integers, you could write:

```
int myNum[3] = {10, 20, 30};
```

## Access the Elements of an Array

You access an array element by referring to the index number inside square brackets `[]`.

This statement accesses the value of the **first element** in **cars**:

### Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

**Note:** Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

## Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

### Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];
// Now outputs Opel instead of Volvo
```

## Loop Through an Array

You can loop through the array elements with the `for` loop.

The following example outputs all elements in the **cars** array:

### Example

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
  cout << cars[i] << "\n";
}
```

This example outputs the index of each element together with its value:

### Example

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
  cout << i << " = " << cars[i] << "\n";
}
```

And this example shows how to loop through an array of integers:

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

## The foreach Loop

There is also a "**for-each** loop" (introduced in C++ version 11 (2011), which is used exclusively to loop through elements in an array:

### Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

The following example outputs all elements in an array, using a "**for-each** loop":

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

## Omit Array Size

In C++, you don't have to specify the size of the array. The compiler is smart enough to determine the size of the array based on the number of inserted values:

```
string cars[] = {"Volvo", "BMW", "Ford"}; // Three array elements
```

The example above is equal to:

```
string cars[3] = {"Volvo", "BMW", "Ford"}; // Also three array elements
```

However, the last approach is considered as "good practice", because it will reduce the chance of errors in your program.

## Omit Elements on Declaration

It is also possible to declare an array without specifying the elements on declaration, and add them later:

### Example

```
string cars[5];
cars[0] = "Volvo";
cars[1] = "BMW";
...
```

## Get the Size of an Array

To get the size of an array, you can use the `sizeof()` operator:

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
cout << sizeof(myNumbers);
```

Result:

```
20
```

Why did the result show `20` instead of `5`, when the array contains 5 elements?

It is because the `sizeof()` operator returns the size of a type in **bytes**.

You learned from the Data Types chapter that an `int` type is usually 4 bytes, so from the example above, *4 x 5 (4 bytes x 5 elements) =* ***20 bytes***.

**To find out how many elements an array has**, you have to divide the size of the array by the size of the data type it contains:

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
int getArrayLength = sizeof(myNumbers) / sizeof(int);
cout << getArrayLength;
```

Result:

```
5
```

## Loop Through an Array with sizeof()

In the Arrays and Loops Chapter, we wrote the size of the array in the loop condition (`i < 5`). This is not ideal, since it will only work for arrays of a specified size.

However, by using the `sizeof()` approach from the example above, we can now make loops that work for arrays of any size, which is more sustainable.

Instead of writing:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

It is better to write:

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < sizeof(myNumbers) / sizeof(int); i++) {
  cout << myNumbers[i] << "\n";
}
```

Note that, in C++ version 11 (2011), you can also use the "for-each" loop:

### Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

It is good to know the different ways to loop through an array, since you may encounter them all in different programs.

## Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays.

To declare a multi-dimensional array, define the variable type, specify the name of the array followed by square brackets which specify how many elements the main array has, followed by another set of square brackets which indicates how many elements the sub-arrays have:

```
string letters[2][4];
```

As with ordinary arrays, you can insert values with an array literal - a comma-separated list inside curly braces. In a multi-dimensional array, each element in an array literal is another array literal.

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
```

Each set of square brackets in an array declaration adds another **dimension** to an array. An array like the one above is said to have two dimensions.

Arrays can have any number of dimensions. The more dimensions an array has, the more complex the code becomes. The following array has three dimensions:

```
string letters[2][2][2] = {
  {
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};
```

## Access the Elements of a Multi-Dimensional Array

To access an element of a multi-dimensional array, specify an index number in each of the array's dimensions.

This statement accesses the value of the element in the **first row (0)** and **third column (2)** of the **letters** array.

### Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};

cout << letters[0][2];  // Outputs "C"
```

**Remember that:** Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

## Change Elements in a Multi-Dimensional Array

To change the value of an element, refer to the index number of the element in each of the dimensions:

### Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
letters[0][0] = "Z";

cout << letters[0][0];  // Now outputs "Z" instead of "A"
```

## Loop Through a Multi-Dimensional Array

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following example outputs all elements in the **letters** array:

### Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 4; j++) {
    cout << letters[i][j] << "\n";
  }
}
```

This example shows how to loop through a three-dimensional array:

### Example

```
string letters[2][2][2] = {
  {
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};
```

```
for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 2; k++) {
      cout << letters[i][j][k] << "\n";
    }
  }
}
```

## Why Multi-Dimensional Arrays?

Multi-dimensional arrays are great at representing grids. This example shows a practical use for them. In the following example we use a multi-dimensional array to represent a small game of Battleship:

### Example

```
// We put "1" to indicate there is a ship.
bool ships[4][4] = {
  { 0, 1, 1, 0 },
  { 0, 0, 0, 0 },
  { 0, 0, 1, 0 },
  { 0, 0, 1, 0 }
};

// Keep track of how many hits the player has and how many turns they have
played in these variables
int hits = 0;
int numberOfTurns = 0;

// Allow the player to keep going until they have hit all four ships
while (hits < 4) {
  int row, column;

  cout << "Selecting coordinates\n";

  // Ask the player for a row
  cout << "Choose a row number between 0 and 3: ";
  cin >> row;

  // Ask the player for a column
  cout << "Choose a column number between 0 and 3: ";
  cin >> column;

  // Check if a ship exists in those coordinates
  if (ships[row][column]) {
    // If the player hit a ship, remove it by setting the value to zero.
    ships[row][column] = 0;

    // Increase the hit counter
    hits++;

    // Tell the player that they have hit a ship and how many ships are left
    cout << "Hit! " << (4-hits) << " left.\n\n";
  } else {
    // Tell the player that they missed
    cout << "Miss\n\n";
  }

  // Count how many turns the player has taken
  numberOfTurns++;
}

cout << "Victory!\n";
cout << "You won in " << numberOfTurns << " turns";
```

## C++ Structures

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

## Create a Structure

To create a structure, use the `struct` keyword and declare each of its members inside curly braces.

After the declaration, specify the name of the structure variable (**myStructure** in the example below):

```
struct {             // Structure declaration
  int myNum;         // Member (int variable)
  string myString;   // Member (string variable)
} myStructure;       // Structure variable
```

## Access Structure Members

To access members of a structure, use the dot syntax (`.`):

### Example

Assign data to members of a structure and print it:

```
// Create a structure variable called myStructure
struct {
  int myNum;
  string myString;
} myStructure;

// Assign values to members of myStructure
```

```
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Print members of myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

## One Structure in Multiple Variables

You can use a comma (`,`) to use one structure in many variables:

```
struct {
  int myNum;
  string myString;
} myStruct1, myStruct2, myStruct3; // Multiple structure variables separated
with commas
```

This example shows how to use a structure in two different variables:

### Example

Use one structure to represent two cars:

```
struct {
  string brand;
  string model;
  int year;
} myCar1, myCar2; // We can add variables by separating them with a comma
here

// Put data into the first structure
myCar1.brand = "BMW";
myCar1.model = "X5";
myCar1.year = 1999;

// Put data into the second structure
myCar2.brand = "Ford";
myCar2.model = "Mustang";
myCar2.year = 1969;

// Print the structure members
cout << myCar1.brand << " " << myCar1.model << " " << myCar1.year << "\n";
cout << myCar2.brand << " " << myCar2.model << " " << myCar2.year << "\n";
```

## Named Structures

By giving a name to the structure, you can treat it as a data type. This means that you can create variables with this structure anywhere in the program at any time.

To create a named structure, put the name of the structure right after the `struct` keyword:

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
  string myString;
};
```

To declare a variable that uses the structure, use the name of the structure as the data type of the variable:

```
myDataType myVar;
```

## Example

Use one structure to represent two cars:

```
// Declare a structure named "car"
struct car {
  string brand;
  string model;
  int year;
};

int main() {
  // Create a car structure and store it in myCar1;
  car myCar1;
  myCar1.brand = "BMW";
  myCar1.model = "X5";
  myCar1.year = 1999;

  // Create another car structure and store it in myCar2;
  car myCar2;
  myCar2.brand = "Ford";
  myCar2.model = "Mustang";
  myCar2.year = 1969;

  // Print the structure members
  cout << myCar1.brand << " " << myCar1.model << " " << myCar1.year << "\n";
  cout << myCar2.brand << " " << myCar2.model << " " << myCar2.year << "\n";

  return 0;
}
```

## Creating References

A reference variable is a "reference" to an existing variable, and it is created with the `&` operator:

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

Now, we can use either the variable name `food` or the reference name `meal` to refer to the `food` variable:

### Example

```
string food = "Pizza";
string &meal = food;

cout << food << "\n";  // Outputs Pizza
cout << meal << "\n";  // Outputs Pizza
```

## Memory Address

In the example from the previous page, the `&` operator was used to create a reference variable. But it can also be used to get the memory address of a variable; which is the location of where the variable is stored on the computer.

When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

To access it, use the `&` operator, and the result will represent where the variable is stored:

### Example

```
string food = "Pizza";

cout << &food; // Outputs 0x6dfed4
```

**Note:** The memory address is in hexadecimal form (0x..). Note that you may not get the same result in your program.

***And why is it useful to know the memory address?***

**References** and **Pointers** (which you will learn about in the next chapter) are important in C++, because they give you the ability to manipulate the data in the computer's memory - **which can reduce the code and improve the performance**.

These two features are one of the things that make C++ stand out from other programming languages, like Python and Java.

## Creating Pointers

You learned from the previous chapter, that we can get the **memory address** of a variable by using the `&` operator:

### Example

```
string food = "Pizza"; // A food variable of type string

cout << food;  // Outputs the value of food (Pizza)
cout << &food; // Outputs the memory address of food (0x6dfed4)
```

A **pointer** however, is a variable that **stores the memory address as its value**.

A pointer variable points to a data type (like `int` or `string`) of the same type, and is created with the `*` operator. The address of the variable you're working with is assigned to the pointer:

## Example

```
string food = "Pizza";  // A food variable of type string
string* ptr = &food;    // A pointer variable, with the name ptr, that stores
the address of food

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";
```

### *Example explained*

Create a pointer variable with the name `ptr`, that **points to** a `string` variable, by using the asterisk sign `*` (`string* ptr`). Note that the type of the pointer has to match the type of the variable you're working with.

Use the `&` operator to store the memory address of the variable called `food`, and assign it to the pointer.

Now, `ptr` holds the value of `food`'s memory address.

**Tip:** There are three ways to declare pointer variables, but the first way is preferred:

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

# Get Memory Address and Value

In the example from the previous page, we used the pointer variable to get the memory address of a variable (used together with the `&` **reference** operator). However, you can also use the pointer to get the value of the variable, by using the `*` operator (the **dereference** operator):

## Example

```
string food = "Pizza";  // Variable declaration
string* ptr = &food;    // Pointer declaration

// Reference: Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";
```

```
// Dereference: Output the value of food with the pointer (Pizza)
cout << *ptr << "\n";
```

Note that the * sign can be confusing here, as it does two different things in our code:

- When used in declaration (string* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

## Modify the Pointer Value

You can also change the pointer's value. But note that this will also change the value of the original variable:

### Example

```
string food = "Pizza";
string* ptr = &food;

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Access the memory address of food and output its value (Pizza)
cout << *ptr << "\n";

// Change the value of the pointer
*ptr = "Hamburger";

// Output the new value of the pointer (Hamburger)
cout << *ptr << "\n";

// Output the new value of the food variable (Hamburger)
cout << food << "\n";
```

# C++ Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

## Create a Function

C++ provides some pre-defined functions, such as `main()`, which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

### Syntax

```
void myFunction() {
  // code to be executed
}
```

### *Example Explained*

- `myFunction()` is the name of the function
- `void` means that the function does not have a return value. You will learn more about return values later in the next chapter
- inside the function (the body), add code that defines what the function should do

## Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`

In the following example, `myFunction()` is used to print a text (the action), when it is called:

### Example

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

### Example

```
void myFunction() {
  cout << "I just got executed!\n";
}

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

// I just got executed!
// I just got executed!
// I just got executed!
```

## Function Declaration and Definition

A C++ function consist of two parts:

- **Declaration:** the return type, the name of the function, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

**Note:** If a user-defined function, such as `myFunction()` is declared after the `main()` function, **an error will occur**:

### Example

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

void myFunction() {
  cout << "I just got executed!";
}

// Error
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above `main()`, and function definition below `main()`. This will make the code better organized and easier to read:

### Example

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

## Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

### Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a `string` called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

### Example

```
void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**,
while `Liam`, `Jenny` and `Anja` are **arguments**.

## Default Parameter Value

You can also use a default parameter value, by using the equals sign (`=`).

If we call the function without an argument, it uses the default value ("Norway"):

### Example

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, `country` is an optional parameter and `"Norway"` is the default value.

## Multiple Parameters

Inside the function, you can add as many parameters as you want:

### Example

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Note that when you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

## Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function:

### Example

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  cout << myFunction(3);
  return 0;
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a function with **two parameters**:

### Example

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

int main() {
  cout << myFunction(5, 3);
  return 0;
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable:

### Example

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

int main() {
  int z = myFunction(5, 3);
  cout << z;
  return 0;
}
// Outputs 8 (5 + 3)
```

## Pass By Reference

In the examples from the previous page, we used normal variables when we passed parameters to a function. You can also pass a reference to the function. This can be useful when you need to change the value of the arguments:

### Example

```
void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 10;
  int secondNum = 20;

  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  // Call the function, which will change the values of firstNum and
secondNum
  swapNums(firstNum, secondNum);

  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

## Pass Arrays as Function Parameters

You can also pass arrays to a function:

### Example

```
void myFunction(int myNumbers[5]) {
  for (int i = 0; i < 5; i++) {
    cout << myNumbers[i] << "\n";
  }
}

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  myFunction(myNumbers);
  return 0;
}
```

***Example Explained***

The function (`myFunction`) takes an array as its parameter (`int myNumbers[5]`), and loops through the array elements with the `for` loop.

When the function is called inside `main()`, we pass along the `myNumbers` array, which outputs the array elements.

**Note** that when you call the function, you only need to use the name of the array when passing it as an argument `myFunction(myNumbers)`. However, the full declaration of the array is needed in the function parameter (`int myNumbers[5]`).

## Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

### Example

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

Consider the following example, which have two functions that add numbers of different type:

### Example

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

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

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 = plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Instead of defining two functions that should do the same thing, it is better to overload one.

In the example below, we overload the `plusFunc` function to work for both `int` and `double`:

### Example

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

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

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

**Note:** Multiple functions can have the same name as long as the number and/or type of parameters are different.

## Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

## Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

### Example

```
int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}

int main() {
  int result = sum(10);
  cout << result;
  return 0;
}
```

## Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Since the function does not call itself when `k` is 0, the program stops there and returns the result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

# C++ What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

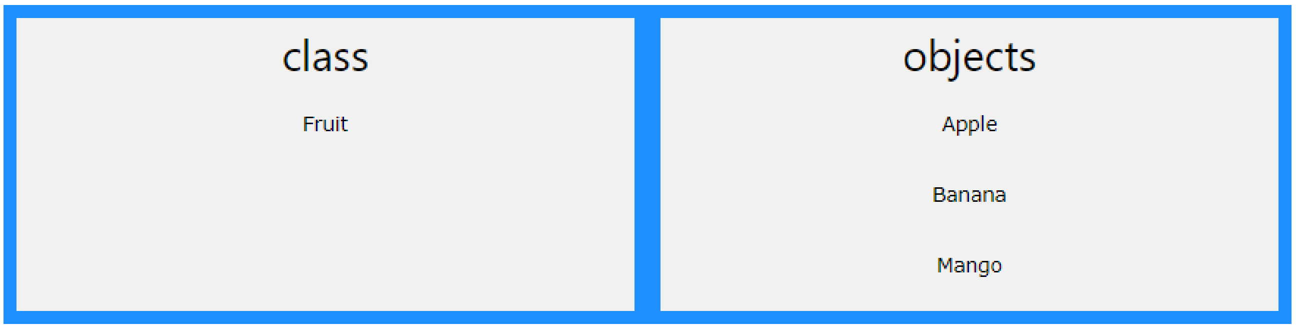
- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

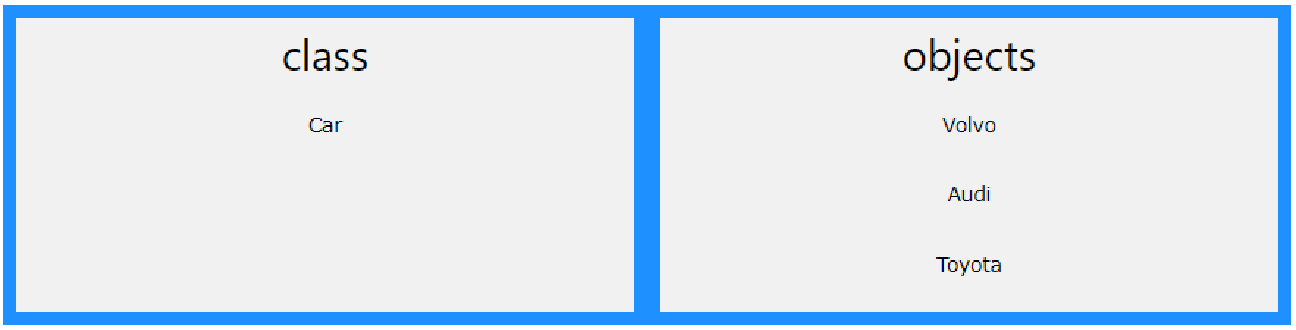
## C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



Another example:



So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and functions from the class.

## C++ Classes/Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the `class` keyword:

## Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

## Example explained

- The class keyword is used to create a class called MyClass.
- The public keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;.

# Create an Object

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

## Example

Create an object called "myObj" and access the attributes:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

## Multiple Objects

You can create multiple objects of one class:

### Example

```
// Create a Car class with some attributes
class Car {
  public:
    string brand;
    string model;
    int year;
};

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

## Class Methods

Methods are **functions** that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

In the following example, we define a function inside the class, and we name it "`myMethod`".

**Note:** You access methods just like you access attributes; by creating an object of the class and using the dot syntax (`.`):

### Inside Example

```
class MyClass {         // The class
  public:               // Access specifier
    void myMethod() {   // Method/function defined inside the class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifiying the name of the class, followed the scope resolution `::` operator, followed by the name of the function:

### Outside Example

```
class MyClass {         // The class
  public:               // Access specifier
    void myMethod();    // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

## Parameters

You can also add parameters:

### Example

```
#include <iostream>
using namespace std;

class Car {
  public:
    int speed(int maxSpeed);
};
```

```
int Car::speed(int maxSpeed) {
  return maxSpeed;
}

int main() {
  Car myObj; // Create an object of Car
  cout << myObj.speed(200); // Call the method with an argument
  return 0;
}
```

## Constructors

A constructor in C++ is a **special method** that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses `()`:

### Example

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the
constructor)
  return 0;
}
```

**Note:** The constructor has the same name as the class, it is always `public`, and it does not have any return value.

## Constructor Parameters

Constructors can also take parameters (just like regular functions), which can be useful for setting initial values for attributes.

The following class have `brand`, `model` and `year` attributes, and a constructor with different parameters. Inside the constructor we set the attributes equal to the constructor parameters (`brand=x`, etc). When we call the constructor (by creating an object of the class), we pass parameters to the constructor, which will set the value of the corresponding attributes to the same:

## Example

```
class Car {          // The class
  public:            // Access specifier
    string brand;    // Attribute
    string model;    // Attribute
    int year;        // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution `::` operator, followed by the name of the constructor (which is the same as the class):

## Example

```
class Car {          // The class
  public:            // Access specifier
    string brand;    // Attribute
    string model;    // Attribute
    int year;        // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);
  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

## Access Specifiers

By now, you are quite familiar with the `public` keyword that appears in all of our class examples:

### Example

```
class MyClass {  // The class
  public:        // Access specifier
    // class members goes here
};
```

The `public` keyword is an **access specifier.** Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are `public` - which means that they can be accessed and modified from outside the code.

However, what if we want members to be private and hidden from the outside world?

In C++, there are three access specifiers:

- `public` - members are accessible from outside the class
- `private` - members cannot be accessed (or viewed) from outside the class
- `protected` - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

In the following example, we demonstrate the differences between `public` and `private` members:

### Example

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

If you try to access a private member, an error occurs:

```
error: y is private
```

**Note:** It is possible to access private members of a class using a public method inside the same class. See the next chapter (Encapsulation) on how to do this.

**Tip:** It is considered good practice to declare your class attributes as private (as often as you can). This will reduce the possibility of yourself (or others) to mess up the code. This is also the main ingredient of the Encapsulation concept, which you will learn more about in the next chapter.

**Note:** By default, all members of a class are `private` if you don't specify an access specifier:

### Example

```
class MyClass {
  int x;   // Private attribute
  int y;   // Private attribute
};
```

## Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as `private` (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

## Access Private Members

To access a private attribute, use public "get" and "set" methods:

### Example

```
#include <iostream>
using namespace std;

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

### *Example explained*

The `salary` attribute is `private`, which have restricted access.

The public `setSalary()` method takes a parameter (`s`) and assigns it to the `salary` attribute (salary = s).

The public `getSalary()` method returns the value of the private `salary` attribute.

Inside `main()`, we create an object of the `Employee` class. Now we can use the `setSalary()` method to set the value of the private attribute to `50000`. Then we call the `getSalary()` method on the object to return the value.

## Why Encapsulation?

- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

## Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

## Types of Abstraction:

1. Data abstraction – This type only shows the required information about the data and hides the unnecessary data.
2. Control Abstraction – This type only shows the required information about the implementation and hides unnecessary information.

## Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

## Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

## Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

Members declared as **public** in a class can be accessed from anywhere in the program.

Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

## Difference between Abstraction and Encapsulation

In OOPs, Abstraction is the method of getting information where the information needed will be taken in such a simplest way that solely the required components are extracted, and also the ones that are considered less significant are unnoticed. The concept of abstraction only shows necessary information to the users. It reduces the complexity of the program by hiding the implementation complexities of programs.

**Example of Abstraction:**

```
#include <iostream>
using namespace std;

class Summation {
private:
        // private variables
        int a, b, c;
public:
        void sum(int x, int y)
        {
                a = x;
                b = y;
                c = a + b;
                cout<<"Sum of the two number is : "<<c<<endl;
        }
};
int main()
{
        Summation s;
        s.sum(5, 4);
        return 0;
}
```

**Output:**

```
Sum of the two number is: 9
```

In the this example, we can see that abstraction has achieved by using class. The class ‘Summation’ holds the private members a, b and c, which are only accessible by the member functions of that class.

Encapsulation is the process or method to contain the information. Encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside world. This method encapsulates the data and function together inside a class which also results in data abstraction.

Example of Encapsulation:

```
#include <iostream>
using namespace std;

class EncapsulationExample {
private:
        // we declare a as private to hide it from outside
        int a;

public:
        // set() function to set the value of a
        void set(int x)
        {
                a = x;
        }

        // get() function to return the value of a
        int get()
        {
                return a;
        }
};

// main function
int main()
{
        EncapsulationExample e1;

        e1.set(10);

        cout<<e1.get();
        return 0;
}
```

**Output:**

```
10
```

In the this program, the variable a is made private so that this variable can be accessed and manipulated only by using the methods get() and set() that are present within the class. Therefore we can say that, the variable a and the methods set() as well as get() have binded together that is nothing but encapsulation.

| S.NO | Abstraction | Encapsulation |
|---|---|---|
| 1. | Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| 2. | In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| 3. | Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| 4. | We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| 5. | In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| 6. | The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |

## Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

In the example below, the `Car` class (child) inherits the attributes and methods from the `Vehicle` class (parent):

## Example

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

***Why And When To Use "Inheritance"?***

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

## Multilevel Inheritance

A class can also be derived from one class, which is already derived from another class.

In the following example, `MyGrandChild` is derived from class `MyChild` (which is derived from `MyClass`).

## Example

```
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};
```

```
// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

## Multiple Inheritance

A class can also be derived from more than one base class, using a **comma-separated list:**

### Example

```
// Base class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Another base class
class MyOtherClass {
  public:
    void myOtherFunction() {
      cout << "Some content in another class." ;
    }
};

// Derived class
class MyChildClass: public MyClass, public MyOtherClass {
};

int main() {
  MyChildClass myObj;
  myObj.myFunction();
  myObj.myOtherFunction();
  return 0;
}
```

## Access Specifiers

You learned from the Access Specifiers chapter that there are three specifiers available in C++. Until now, we have only used `public` (members of a class are accessible from outside the class) and `private` (members can only be accessed within the class). The third specifier, `protected`, is similar to `private`, but it can also be accessed in the **inherited** class:

## Example

```
// Base class
class Employee {
  protected: // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
  public:
    int bonus;
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Programmer myObj;
  myObj.setSalary(50000);
  myObj.bonus = 15000;
  cout << "Salary: " << myObj.getSalary() << "\n";
  cout << "Bonus: " << myObj.bonus << "\n";
  return 0;
}
```

# Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

## Example

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
```

```
// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
```

Remember from the Inheritance chapter that we use the `:` symbol to inherit from a class.

Now we can create `Pig` and `Dog` objects and override the `animalSound()` method:

## Example

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

***Why And When To Use "Inheritance" and "Polymorphism"?***

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

## C++ Files

The `fstream` library allows us to work with files.

To use the `fstream` library, include both the standard `<iostream>` **AND** the `<fstream>` header file:

### Example

```
#include <iostream>
#include <fstream>
```

There are three classes included in the `fstream` library, which are used to create, write or read files:

| Class | Description |
|---|---|
| `ofstream` | Creates and writes to files |
| `ifstream` | Reads from files |
| `fstream` | A combination of ofstream and ifstream: creates, reads, and writes to files |

## Create and Write To a File

To create a file, use either the `ofstream` or `fstream` class, and specify the name of the file.

To write to the file, use the insertion operator (`<<`).

### Example

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  // Create and open a text file
  ofstream MyFile("filename.txt");

  // Write to the file
  MyFile << "Files can be tricky, but it is fun enough!";
  // Close the file
  MyFile.close();
}
```

***Why do we close the file?***

It is considered good practice, and it can clean up unnecessary memory space.

## Read a File

To read from a file, use either the `ifstream` or `fstream` class, and the name of the file.

Note that we also use a `while` loop together with the `getline()` function (which belongs to the `ifstream` class) to read the file line by line, and to print the content of the file:

### Example

```
// Create a text string, which is used to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// Use a while loop together with the getline() function to read the file
line by line
while (getline (MyReadFile, myText)) {
  // Output the text from the file
  cout << myText;
}

// Close the file
MyReadFile.close();
```

## C++ Exceptions

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

## C++ try and catch

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

## Example

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

Consider the following example:

## Example

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

### *Example explained*

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it.
The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

## Example

```
int age = 20;
```

You can also use the throw keyword to output a reference number, like a custom error number/code for organizing purposes:

## Example

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Error number: " << myNum;
}
```

## Handle Any Type of Exceptions (...)

If you do not know the `throw` **type** used in the `try` block, you can use the "three dots" syntax (`...`) inside the `catch` block, which will handle any type of exception:

## Example

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least 18 years old.\n";
}
```

## Operator Overloading

In C++, Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

In this article, we will further discuss about operator overloading in C++ with examples and see which operators we can or cannot overload in C++.

**Implementation:**

```
// C++ Program to Demonstrate the

// working/Logic behind Operator

// Overloading

class A {

      statements;
};
int main()
{
      A a1, a2, a3;
      a3 = a1 + a2;
      return 0;
}
```

In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

Now, if the user wants to make the operator “+” add two class objects, the user has to redefine the meaning of the “+” operator such that it adds two class objects. This is done by using the concept of “Operator overloading”. So the main idea behind “Operator overloading” is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

### 1. Overloading Unary Operator

Let us consider overloading (-) unary operator. In the unary operator function, no arguments should be passed. It works only with one class object. It is the overloading of an operator operating on a single operand.

Example: Assume that class Distance takes two member objects i.e. feet and inches, and creates a function by which the Distance object should decrement the value of feet and inches by 1 (having a single operand of Distance Type).

```
// C++ program to show unary
// operator overloading
#include <iostream>
using namespace std;

class Distance {
public:
    int feet, inch;

    // Constructor to initialize
    // the object's value
    Distance(int f, int i)

    {

        this->feet = f;

        this->inch = i;

    }


    // Overloading(-) operator to

    // perform decrement operation

    // of Distance object

    void operator-()

    {

        feet--;

        inch--;

        cout << "\n Feet & Inches(Decrement): " <<

                  feet << "'" << inch;

    }

};


// Driver Code

int main()

{

    Distance d1(8, 9);


    // Use (-) unary operator by

    // single operand

    -d1;

    return 0;

}
```

**Output**

```
Feet & Inches(Decrement): 7'8
```

**Note:** d2 = -d1 will not work, because operator-() does not return any value.

### 2. Overloading Binary Operator

In the binary operator overloading function, there should be one argument to be passed. It is the overloading of an operator operating on two operands. Below is the C++ program to show the overloading of the binary operator (+) using a class Distance with two distant objects.

```
// C++ program to show binary
// operator overloading
#include <iostream>
using namespace std;

class Distance {
public:
    int feet, inch;

    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }

    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }

    // Overloading (+) operator to
    // perform addition of two distance
    // object
    // Call by reference
    Distance operator+(Distance& d2)
    {
        // Create an object to return
        Distance d3;



        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
};

// Driver Code
int main()
{
    Distance d1(8, 9);
    Distance d2(10, 2);
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

    cout << "\nTotal Feet & Inches: " <<
             d3.feet << "'" << d3.inch;
    return 0;
}
```

**Output**

```
Total Feet & Inches: 18'11
```

**Criteria/Rules to Define the Operator Function**

1. In the case of a **non-static member function**, the binary operator should have only one argument and the unary should not have an argument.
2. In the case of a **friend function**, the binary operator should have only two arguments and the unary should have only one argument.
3. Operators that cannot be overloaded are **.* :: ?:**
4. Operators that cannot be overloaded when declaring that function as friend function are **= () [] ->**.
5. The operator function must be either a non-static (member function), global free function or a friend function.

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

sizeof

typeid

Scope resolution (::)

Class member access operators (.(dot), .* (pointer to member operator))

Ternary or conditional (?:)

**Difference between Operator Functions and Normal Functions**

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.