C++ is a popular programming language.C++ is used to create computer programs, and is one of the most used language in game development.

## **Examples in Each Chapter**

Our "Try it Yourself" editor makes it easy to learn C++. You can edit C++ code and view the result in your browser.

### Example

#include <iostream>  
using namespace std;  
int main() {  
cout <<"Hello World!";  
return 0;  
}

We recommend reading this tutorial, in the sequence listed in the left menu. C++ is an object oriented language and some concepts may be new. Take breaks when needed, and go over the examples as many times as needed.

# **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](https://www.w3schools.com/c/index.php).

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#](https://www.w3schools.com/cs/index.php) and [Java](https://www.w3schools.com/java/default.asp), 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](https://www.w3schools.com/c/index.php), 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.

# **C++ Getting Started**

## **C++ Get Started**

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/](https://www.codeblocks.org/downloads/binaries/). 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

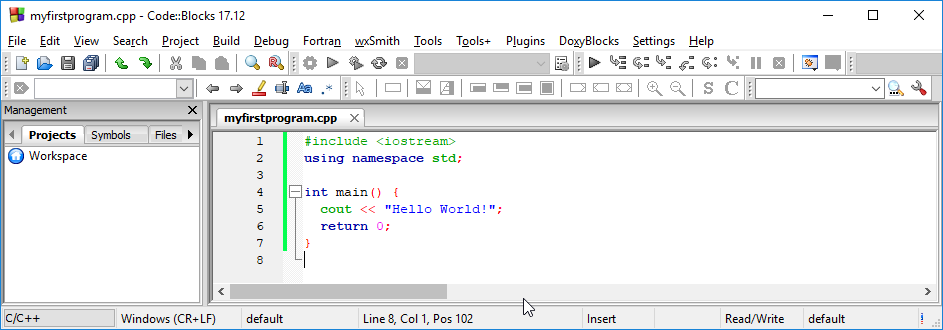
(**File > Save File as**):

#### myfirstprogram.cpp

#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:



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.

## **Learning C++**

When learning C++ at W3Schools.com, you can use our "Try it Yourself" tool, which shows both the code and the result. This will make it easier for you to understand every part as we move forward:

### myfirstprogram.cpp

Code:

#include <iostream>  
using namespace std;  
int main() {  
  cout << "Hello World!";  
  return 0;  
}

Result:

Hello World!

# **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)**

## **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;  
}

# **C++ New Lines**

## **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**

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).

This example uses a single-line comment before a line of code:

### **Example**

// This is a comment  
cout <<"Hello World!";

[Try it Yourself »](https://www.w3schools.com/cpp/trycpp.asp?filename=demo_single_comment)

This example uses a single-line comment at the end of a line of code:

### **Example**

cout <<"Hello World!";// This is a comment[Try it Yourself »](https://www.w3schools.com/cpp/trycpp.asp?filename=demo_single_comment_end)

## **C++ Multi-line Comments**

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

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

### **Example**

/\* The code below will print the words Hello World!  
to the screen, and it is amazing \*/  
cout <<"Hello World!";

#### Single or multi-line comments?

It is up to you which you want to use. Normally, we use // for short comments, and /\* \*/ for longer.

## **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)

You will learn more about the individual types in the Data Types chapter.

## **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.";

## **Add Variables Together**

To add a variable to another variable, you can use the + operator:

### **Example**

int x = 5;  
int y = 6;  
int sum = x + y;  
cout << sum;

# **C++ Declare Multiple Variables**

## **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;

## **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

# **C++ Constants**

## **Constants**

When you do not want others (or yourself) to override 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;

# **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;

There you go! You just built a basic calculator!

# **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 7 decimal digits |
| double | 8 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits |

You will learn more about the individual data types in the next chapters.

# **C++ Numeric Data Types**

## **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.

### **Scientific Numbers**

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

### **Example**

float f1 = 35e3;  
double d1 = 12E4;  
cout << f1;  
cout << d1;

# **C++ Boolean Data Types**

## **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)

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

# **C++ Character Data Types**

## **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, you can use ASCII values to display certain characters:

### **Example**

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

**Tip:** A list of all ASCII values can be found in our ASCII Table Reference.

# **C++ String Data Types**

## **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**

Operators are used to perform operations on variables and values.

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:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## **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 |

# **C++ Assignment Operators**

## **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 |

# **C++ Comparison Operators**

## **Comparison Operators**

Comparison operators are used to compare two values.

**Note:** The return value of a comparison is either true (1) or false (0).

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 |

You will learn much more about comparison operators and how to use them in a later chapter.

# **C++ 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) |

# **C++ Strings**

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes:

### **Example**

Create a variable of type string and assign it a value:

string greeting = "Hello";

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";

## **C++ 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;

## **C++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;

# **C++ 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();

# **C++ 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

## **C++User Input Strings**

It is possible to use the extraction operator >> on cin to display 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 display 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++ String Namespace**

## **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 string (and cout) objects:

### **Example**

#include <iostream>  
#include <string>  
  
int main() {  
  **std::**string greeting = "Hello";  
  **std::**cout << greeting;  
  return 0;  
}

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

## **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);

## **Other Math Functions**

A list of other popular Math functions (from the <cmath> library) can be found in the table below:

|  |  |
| --- | --- |
| Function | Description |
| abs(x) | Returns the absolute value of x |
| acos(x) | Returns the arccosine of x |
| asin(x) | Returns the arcsine of x |
| atan(x) | Returns the arctangent of x |
| cbrt(x) | Returns the cube root of x |
| ceil(x) | Returns the value of x rounded up to its nearest integer |
| cos(x) | Returns the cosine of x |
| cosh(x) | Returns the hyperbolic cosine of x |
| exp(x) | Returns the value of Ex |
| expm1(x) | Returns ex -1 |
| fabs(x) | Returns the absolute value of a floating x |
| fdim(x, y) | Returns the positive difference between x and y |
| floor(x) | Returns the value of x rounded down to its nearest integer |
| hypot(x, y) | Returns sqrt(x2 +y2) without intermediate overflow or underflow |
| 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 |
| pow(x, y) | Returns the value of x to the power of y |
| sin(x) | Returns the sine of x (x is in radians) |
| sinh(x) | Returns the hyperbolic sine of a double value |
| tan(x) | Returns the tangent of an angle |
| tanh(x) | Returns the hyperbolic tangent of a double value |

# **C++ Booleans**

Very often, in programming, you will need a data type that can only have one of two values, like:

* YES / NO
* ON / OFF
* TRUE / FALSE

For this, C++ has a bool data type, which can take the values true (1) or false (0).

## **Boolean Values**

A boolean variable is declared with the bool keyword and can only take the values true or false:

### **Example**

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

From the example above, you can read that a true value returns 1, and false returns 0.

However, it is more common to return boolean values from boolean expressions (see next page).

# **C++ Boolean Expressions**

A **Boolean expression** is a C++ expression that returns a boolean value: 1 (true) or 0 (false).

You can use a comparison operator, such as the **greater than** (>) operator to find out if an expression (or a variable) is true:

### **Example**

int x = 10;  
int y = 9;  
cout << (x > y); // returns 1 (true), because 10 is higher than 9

Or even easier:

### **Example**

cout << (10 > 9); // returns 1 (true), because 10 is higher than 9

In the examples below, we use the **equal to** (==) operator to evaluate an expression:

### **Example**

int x = 10;  
cout << (x == 10);  // returns 1 (true), because the value of x is equal to 10

### **Example**

cout << (10 == 15);  // returns 0 (false), because 10 is not equal to 15

Booleans are the basis for all C++ comparisons and conditions.

You will learn more about conditions (if...else) in the next chapter.

## **C++ Conditions and If Statements**

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";  
}

#### **Example explained**

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "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."

#### **Example explained**

In the example above, time (20) is greater than 18, so the condition is false. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

## **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."

#### **Example explained**

In the example above, time (22) is greater than 10, so the **first condition** is false. The next condition, in the else if statement, is also false, so we move on to the else condition since **condition1** and **condition2** is both false and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

## **C++ Short Hand If Else**

## **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.

## **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++ While Loop**

## **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++;  
}

## **C++ 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);

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

## **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";  
}

# C++ Break and Continue

## **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";  
}

## **Break and Continue in While Loop**

You can also use break and continue in while loops:

### **Break Example**

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

### **Continue Example**

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

# **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

## **C++ Arrays and Loops**

## **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[4] = {"Volvo", "BMW", "Ford", "Mazda"};  
for (int i = 0; i < 4; i++) {  
  cout << cars[i] << "\n";  
}

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

### **Example**

string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};  
for (int i = 0; i < 4; i++) {  
  cout << i << ": " << cars[i] << "\n";  
}

## **C++ Omit Array Size**

## **Omit Array Size**

You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it:

string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3

This is completely fine. However, the problem arise if you want extra space for future elements. Then you have to overwrite the existing values:

~~string cars[] = {"Volvo", "BMW", "Ford"};~~  
string cars[] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};

If you specify the size however, the array will reserve the extra space:

string cars[5] = {"Volvo", "BMW", "Ford"}; // size of array is 5, even though it's only three elements inside it

Now you can add a fourth and fifth element without overwriting the others:

### **Example**

cars[3] = "Mazda";  
cars[4] = "Tesla";

## **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";  
...

## **C++ Array Size**

## **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

# **C++ Multi-Dimensional Arrays**

## **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 (struct)**

## **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;  
}

# **C++ References**

## **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

## **C++ Memory Address**

## **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.