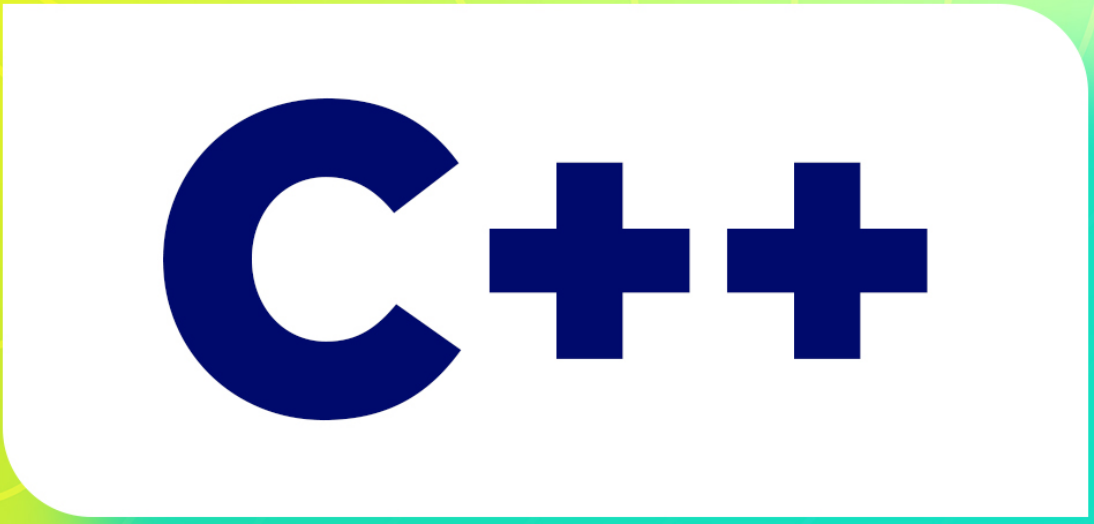
The Ultimate



Part 1: The Fundamentals

This short note is from 1 Hour free video course available on Youtube
Self learning purpose added some changes
Jaydeep Shah (radhey04ec@gmail.com)

Mosh Hamedani

codewithmosh.com

Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Ultimate C++ course** where you will learn everything you need to know from the absolute basics to more advanced concepts. You can find the full course on my website.

https://codewithmosh.com

https://www.youtube.com/c/programmingwithmosh

https://twitter.com/moshhamedani

https://www.facebook.com/programmingwithmosh/

Table of Content

Getting Started……..4

The Basics……..6

Data Types……8

Decision Making……11

Loops……15

Functions……17

Getting Started

Terms

C++ Standard Library
Compiling
Console application
Function
Graphical User Interface (GUI)
Integrated Development Environment
Machine code
Statement
Syntax
Terminal

Summary

- C++ is one of the oldest yet most popular programming languages in the world due to its performance and efficiency.
- It's often used in building performance-critical applications, video games (especially with Unreal Engine), servers, operating systems, etc.
- To learn C++, you need to learn the syntax (grammar) of the language as well as C++ Standard Library, which is a collection of pre-written C++ code for solving common problems.
- To write C++ applications, we often use an Integrated Development Environment (IDE). The most popular IDEs are MS Visual Studio, XCode, and CLion.
- To run C++ applications, first we have to compile our C++ code to machine code.
- The main() function is the starting point of a C++ program.

Your First C++ Program

```cpp
#include <iostream>

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

The Basics

Terms

Camel case
Comment
Constant
Directive
Expression
Hungarian notation
Mathematical expression
Operand
Operator
Pascal case
Snake case
Standard input stream
Standard output stream
Stream extraction operator
Stream insertion operator
Variable

Summary

- We use *variables* to temporarily store data in the computer's memory.
- To declare a variable, we should specify its type and give it a meaningful name.
- We should initialize variables before using them. Using an uninitialized variable can lead to unexpected behavior in our programs since these variables hold garbage values.
- Unlike variables, the value of *constants* don't change.
- The common naming conventions used in C++ applications are: **PascalCase**, **camelCase**, and **snake_case**.
- An *expression* is a piece of code that produces a value. A mathematical (arithmetic) expression consists of an operator (+, -, *, /, %) and two operands.
- Multiplication and division operators have a higher priority than addition and subtraction operators. So, they're applied first.

- We can use parentheses to change the order of operators.
- We use **cout** (pronounced sea-out) to write characters to the *Standard Output Stream* which represents the *terminal* or *console* window.
- We use **cin** (pronounced sea-in) to read data from the *Standard Input Stream* which represents the keyboard.
- We use the *Stream Insertion Operator* (<<) to write data to a stream.
- We use the *Stream Extraction operator* (>>) to read data from a stream.
- The *Standard Template Library* (STL) consists of several files each containing functions for different purposes.
- To use functions in the Standard Library, we should include the corresponding files using the **#include** *directive*.
- Using *comments* we can explain what cannot be expressed in code. This includes why's, how's, and any assumptions we made while writing code.

```
// Declaring a variable
int number = 1;

// Declaring a constant
const double pi = 3.14;

// Mathematical expressions
int x = 10 + 3;

// Writing to the console
cout << "x = " << x;

// Reading from the console
cin >> number;
```


Long and Short effect on datatype

```
//Short and long is useful to increase and decrease range of integer
//C originally written for working on 16 bit machine, but with change of HW architecture there were requirement of size modification in datatype
//short - 16 bit or two byte long atleast, long is 4 byte long (but actual size mostly depends on compiler)

int a;                //Size = 4 bytes
long int b;           //Size = 4 bytes
long long int c;      //Size = 8 bytes
short int d;          //Size = 2 bytes
long e;               //Size = 4 bytes
//short short int e;  Not Allow further shrinking
```

Data Types

Terms

Array
Binary system
Boolean values
Casting
Characters
Compile-time error
Data type
Decimal system
Floating-point number
Hexadecimal system
Index
Overflow
Run-time error
Stream manipulator
String
Underflow

Summary

- C++ has several built-in *data types* for storing *integers* (whole numbers), *floating-point numbers* (numbers with a decimal point), characters, and Boolean values (true/false).
- Floating-point numbers are interpreted as **double** by default. To represent a **float**, we have to add the **F** suffix to our numbers (eg 1.2F).
- Whole numbers are interpreted as **int** by default. To represent a **long,** we have to use the **L** suffix (eg 10L).
- Using the **auto** keyword, we can let the compiler infer the type of a variable based on its initial value.
- Numbers can be represented using the *decimal*, *binary*, and *hexadecimal systems*.
- If we store a value larger or smaller than a data type's limits, *overflowing* or *underflowing* occurs.

- Using the **sizeof()** function, we can see the number of bytes taken by a data type.
- We can use *stream manipulators* to format data sent to a stream. The most common manipulators are **setw, fixed, setprecision, boolalpha, left,** and **right.**
- The Boolean **false** is represented as 0. Any non-zero number is interpreted as the Boolean **true**.
- In C++, characters should be surrounded with single quotes.
- Characters are internally represented as numbers.
- A *string* is a sequence of characters and should be surrounded by double quotes.
- We use *arrays* to store a sequence of items (eg numbers, characters, etc).
- Array elements can be accessed using an *index*. The index of the first element in an array is 0.
- When we store a smaller value in a larger data type, the value gets automatically *cast* (converted to) the larger type. When storing a large value in a smaller data type, we have to explicit cast the value.
- C-style casting involves prefixing a variable with the target data type in parentheses. In C++, we use the **static_cast** operator.
- C++ casting is safer because conversion problems can be caught at the *compile-time*. With C-style casting, we won't know about conversion issues until the *run-time*.


```cpp
// Data types
double price = 9.99;
float interestRate = 3.67F;
long fileSize = 90000L;
char letter = 'a';
string name = "Mosh";
bool isValid = true;
auto years = 5;

// Number systems
int x = 255;
int y = 0b111111;
int z = 0xFF;

// Data types size and limits
int bytes = sizeof(int);
int min = numeric_limits<int>::min();
int max = numeric_limits<int>::max();

// Arrays
int numbers[] = { 1, 2, 3 };
cout << numbers[0];

// C-style casting
double a = 2.0;
int b = (int) a;

// C++ casting
int c = static_cast<int>(a);
```

Decimal / Binary /Hex

For Array return total bytes
int a[10];
cout << sizeof(a); ANSWER = 40 Bytes

Return number of bytes hold by datatype

Useful for finding possible min and max value which can be hold by this datatype , note: need to include <limits> for this

TYPE Casting

Important

Decision Making

Terms

Boolean expression
Comparison operators
Conditional operator
If statement
Logical operators
Nesting if statements
Switch statement

Summary

- We use *comparison operators* to compare values.
- A *Boolean expression* is a piece of code that produces a Boolean value.
- With *Logical operators* we can combine Boolean expressions and represent more complex conditions.
- With the *logical AND operator* (&&) both operands should be true. If either of them is false, the result will be false.
- With the *logical OR operator* (||), the result is true if either of the operands is true.
- The *logical NOT operator* (!) reverses a Boolean value.
- Using ***if*** and ***switch*** *statements*, we can control the logic of our programs.
- An if statement can have zero or more **else if** clauses for evaluating additional conditions.

- An if statement can optionally have an **else** clause.
- We can code an if statement within another. This is called *nesting* if statements.
- Using the *conditional operator* we can simplify many of the if/else statements.
- We use **switch** statements to compare a variable against different values.
- A switch block often has two or more *case labels* and optionally a *default label*.
- Case labels should be terminated with a **break** statement; otherwise, the control moves to the following case label.
- Switch statements are not as flexible as if statements but sometimes they can make our code easier to read.


```
// Comparison operators
bool a = 10 > 5;
bool b = 10 == 10;
bool c = 10 != 5;
```

```
// Logical operators
bool d = a && b; // Logical AND
bool e = a || b; // Logical OR
bool f = !a;     // Logical NOT
```

```
int x = 10;
int y =  !x;
cout << y;
```

Output is 0

```
if (temperature < 60) {
    // ...
}
else if (temperature < 90) {
    // ...
}
else {
    // ...
}
```

```
// Conditional operator
double commission = (sales < 10'000) ? .05 : .1;
```

```
switch (menu) {
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    default:
        // ...
}
```


Loops

Terms

Break statement
Continue statement
Do-while statement
For statement
Infinite loop
Iteration
Loops
Loop variable
Nested loop
While statement

Summary

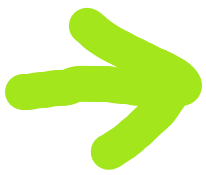
- We use *loops* to repeat a set of statements.
- In C++, we have four types of loops: **for** loops, range-based **for** loops, **while** loops, and **do-while** loops.
- **For** loops are useful when we know ahead of time how many times we want to repeat something.
- Range-based **for** loops are useful when iterating over a list of items (eg an array or a string).
- **While** and **do-while** loops are often used when we don't know ahead of time how many times we need to repeat something.
- Using the **break** statement we can break out of a loop.
- Using the **continue** statement we can skip an iteration.


```cpp
// For loop
for (int i = 0; i < 5; i++)
    cout << i << endl;

// Same algorithm using a while loop
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}

// Same algorithm using a do-while loop
i = 0;
do {
    cout << i << endl;
    i++;
} while (i < 5);
```

```cpp
// Range-based for loop
int numbers[] = { 1, 2, 3 };
for (int number: numbers)
    cout << number << endl;
```

Note : Range based for loop not support in old C++ compiler (C98 etc)

```cpp
int main()
{
 int numbers[] = { 1,11,12};
 for (int a : numbers)
 {
  cout << a << endl;
 }

 string fruit[] = { "apple","banana" };
 for (string b : fruit)
 {
  cout << endl << b;
 }
}
```



OUTPUT:
1
11
12
apple
banana

Functions

Terms

Debugging
Functions
Function arguments
Function declaration
Function definition
Function parameters
Function prototype
Function signature
Global variables
Invoking a function
Header files
Local variables
Namespaces
Overloading functions

Summary

- A *function* is a group of one or more statements that perform a task. Each function should have a clear responsibility. It should do one and only one thing.
- A function can have zero or more *parameters*
- *Arguments* are the values passed to a function.
- To *call* (or *invoke*) a function, we type its name, followed by parenthesis, and the arguments (if any).
- Function parameters can have a default value. This way, we don't have to provide arguments for them.
- The *signature* of a function includes the function name, and the number, order, and type of parameters.

- *Overloading* a function means creating another variation with a different signature. By overloading functions, we can call our functions in different ways.

- Arguments of a function can be passed by value or reference. When passed by value, they get copied to the parameters of the function.

- To pass an argument by a reference, we should add an & after the parameter type.

- *Local variables* are only accessible within the function in which they are defined. *Global variables* are accessible to all functions.

- Global variables can lead to hard-to-detect bugs and should be avoided as much as possible.

- A *function declaration* (also called a *function prototype*) tells the compiler about the existence of a function with a given signature. A *function definition* (or implementation) provides the actual body (or code) for the function.

- As our programs grow in more complexity, it becomes critical to split our code into separate files.

- A *header file* ends with "**.h**" or "**.hpp**" extension and consists of function declarations and constants. We can import header files using the **#include** directive.

- An *implementation file* ends with "**.cpp**" extension and consists of function definitions.

- Using *namespaces* we can prevent name collisions in our programs.

- *Debugging* is a technique for executing a program line by line and identifying potential errors.


```cpp
// Defining functions
void greet(string name) {
    cout << "Hello " << name;
}

string fullName(string firstName, string lastName) {
    return firstName + " " + lastName;
}


// Parameters with a default value
double calculateTax(double income, double taxRate = .3) {
    return income * taxRate;
}


// Overloading functions
void greet(string name) {
}

void greet(string title, string name) {
}

// Reference parameters
void increase(double& number) {
    number++;
}
```



```
// Function declaration
void greet(string name);


// Defining a namespace
namespace messaging {
    void greet(string name) {}
}


// Using a namespace
using namespace messaging;
// or
using messaging::greet;
```

