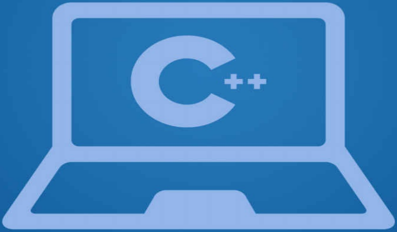
C++

THE NO B.S. C++ CRASH COURSE FOR NEWBIES



LEARN C++ PROGRAMMING IN 8 HOURS!

STEVEN CODEY

C++: The No B.S. C++ Crash Course For Newbies

Learn C++ Programming In 8 Hours!

Introduction

I want to thank you and congratulate you for downloading the book, *"C++: The No B.S. C++ Crash Course for Newbies – Learn C++ Programming in 8 Hours!"*

A century ago, education was all about learning the alphabetical letters. The demand for skills in IT and programming was also very low. Over the years, technology has advanced so much that the world can't do without it.

The internet has changed the world, revolutionizing it into a small well-connected village with information traversing different continents in seconds. Today, people easily access information from the web. Businesses rely on the internet for a range of benefits spanning increased access to a wider audience to being available 24 hours a day and 7 days a week all-year-round.

With almost everyone on the go, mobile devices have become a necessity. Furthermore, more and more people use their smartphones to access the internet. Websites are the core of the web, providing the platforms through which the globe is connected.

In the future, almost everything will depend on technology for automation. Since programming is at the core of these developments, the demand for programmers is not just high, but is expected to blow up in the future.

By 2020, according to a recent study, there'll be need for about 1.4 million developers. However, only 400,000 developers will be available come 2020, indicating an impending shortage of programmers in the industry.

Therefore, learning and developing programming skills is inevitable and this is where you come in. Learning C++ programming language is a good start. This language will give you the foundation you need to become a successful programmer.

This book contains proven steps and strategies on how to code in C++ programming language. This book covers 8 chapters. I will cover various basic yet critical concepts you need to learn as you start your journey towards being an expert programmer. You made the right choice!

Thanks again for downloading this book, I hope you enjoy it! Have lots of fun!

Table of Contents

Conclusion

Chapter 1

The Origin of C++

C++ has its origin dating back to 1979. This was a time when an employee at Bell AT & T, Bjarne Stroustrup, began working on C programming language and its classes. Bjarne borrowed various desirable features from other languages such as CLU, Simula, ML, ALGOL 68 and Ada.

This means that C++ is made up of C programming language features, including strong type checking, basic inheritance, classes and default function argument. Initially known as C with classes, it got its name C++ in 1983. The language specification standards for C++ were released by ANSI-ISO in

1998.

A new standard for C++ known as C++11 was released in mid 2011. It came about as a result of a major influence from the Boost libraries. It ’ s from these libraries that C++ sourced many of its new corresponding modules.

Some of the features sourced from these libraries include atomics support, a comprehensive randomization library, container classes, auto keyword, a standard threading library, regular expression support, array-initialization lists, improved unions support, a new C++ time library and new templates.

In December 2014, C++14 was released with minor bug fixes and enhancements over C++11.

Why Learn C++

Although C++ is a language with its origins in the 80s, it ’ s still a popular programming language that keeps evolving to meet modern developer requirements. There are five major reasons why C++ is an important language hence worth your time and money as a programmer.

- High Portability – C++ is highly portable. This means that you can use it to develop multi-platform and multi-device applications.

- Features a Function Library – C++ has a library rich in functions you’ll need when developing programs.

- Object-Oriented – C++ is an object oriented language for programming. This simply means that it features inheritance, data abstraction, classes, encapsulation and polymorphism.

- Efficient, Powerful and Fast – C++ is not just efficient and powerful, but also a powerful programming language. Use it to develop various programs or applications, ranging from games and 3D graphics to real-time mathematical simulations and GUI apps. It comes in handy when you need direct access to your hardware, and efficient and fast program execution.

Hobbyists have developed open-source games using C++ (wholly or partially). For instance, whereas openTTD was developed using C++, Raspberry Pi Units and Arduino circuits are programmed in C++ and C.

The language is readable and highly functional, enabling you to do anything you want, but within the capability of a language.

- Functions Overloading & Exception Handling – Unlike C programming language, you can easily overload functions and handle exceptions in C++.

So, what ’ s C++?

C++ is pronounced as ‘ C plus-plus ’ . It ’ s an object-oriented programming language for general purposes. As mentioned earlier, it ’ s founded on C programming language and was started by Bjarne Stroustrups with its first release in 1983.

Many applications were developed using C++, including Microsoft applications, part of Mac OS/X, Adobe and MongoDB databases, among other programs or apps. C++ is the best programming language when it comes to developing apps whose performance is critical such as processing of audio/video or development of ‘ twitch ’ games.

App development in C++ continues to benefit businesses, both small and large, worldwide depending on their goals. With constant updates, the range of apps or programs you can develop using C++ continues to grow.

C++ is the most widely used language for programming. However, its popularity slightly dropped over the years due the increasing prevalence of natural and less esoteric languages such as C# and Java. Ruby, Python and Perl scripting languages have also become more prevalent, dropping C++ down the rank.

C++ is still an important low-level programming language. Since the language is universal, learning it can help you maintain its legacy codes.

As a beginner developer, learning C++ is no doubt a good foundation for your programming career. It bridges higher-level and lower-level languages such as Python or Java and C, respectively.

With this knowledge about C++ and why you need to learn it, you can now graduate to the next chapter. Get started and write your first C++ program.

Chapter 2

Getting Started – Write Your First Program in C++

The instructions I ’ ve provided in this eBook are pretty straightforward because I ’ m aware you ’ re a beginner. I ’ m setting you up with this simple guide to enable you start coding as soon as possible. I will give you all the information you need to get your programming career started.

Choose a Compiler and IDE to Use

You need a compiler and IDE to get started with C++ as a novice or beginner. An integrated development environment (IDE) is a platform on which you write your codes. Depending on the IDE you choose, you can perform various functions on the coding environment,ranging from debugging your programs to running the codes you ’ ve written, among other roles.

Humans and machines understand different languages. Whereas humans use natural languages based on words, machines (devices or hardware) comprehend machine language based on codes. Therefore, it ’ s important to convert your written programs into a language that machines can understand.

This is where compilers come in. A compiler is a translator, playing the role of translating the programs you write into a machine language (codes), a form machines can easily read and comprehend.

In many cases, IDEs come with in-built compilers. However, IDEs aren ’ t necessary in writing your programs. This is attributed to the fact that you can use a text editor to theoretically write your code, save it as a .cpp file and run it on a compiler.However, I wouldn ’ t advice you, a beginner, or any other programmer, irrespective of the coding language in question, to use text editors in place of IDEs.

We ’ ll use CodeBlocks IDE for this beginner guide. However, feel free to use any other IDE of your liking as you practice your programming skills. If you wish to use a real-time web-based compiler, consider Rextester.com. Otherwise, visit www.codeblocks.org to access this integrated development environment.

Locate the page for ‘ downloads ’ , and click on the link that reads ‘ download the binary release ’ . Pick the rightversion for your operating system. If you ’ re using Windows, look out for the file titled ‘ codeblocks-(version written here) mingw-setup.exe ’ . Do the same for Mac OS/X, Linux or any other operating system.

Access the executable file in the ‘ Downloads ’ folder and double click on it. Click OK to run the file.Launch the program upon successful installation and select the compiler you ’ d like to use. Note that MinGW/GCC compiler is selected by default. Since we ’ re going to use it, simply clickon the ‘ ok ’ button.

With everything regarding the IDE program set, and its window still active on your computer ’ s desktop, locate ‘ file ’ on the taskbar and click on it. Hover your mouse over it and select the ‘ New ’ sub-menu that pops up. Finally, select the ‘ empty file ’ option.

Although you might not understand this now, you will by the time you finish reading and practicing using this guide. Type the following program into the ‘ empty file ’ you just opened.

```
        #include <iostream>
        using namespace std;
        int main()
                        {
        cout << ” Hello World! I ’ m Here
Now\n ” ;
        return 0;
        }
```

When done, locate the icon bar just below the taskbar on the same IDE window. It has a set of icons including a play, a gear and combinational play/gear buttons. The play button will enable you to run your programs, the gear button is what you need to compile your code and the combinational play/gear button performs both functions.

Note that in C++, you must first compile your program before running or executing it.

Play around with each button and see what happens to the code. Click on the gear button and see what happens. Do the same with the other two icon buttons.

And there you go! That was just your first C++ program. Hold down the ctrl key and hit letter S to save the file or click on the ‘ Edit ’ menu
then ‘ Save ’ sub-menu to do the same. Name the file as ‘ FirstCplusplus.cpp ’ .

Watch what happens to your first C++ code.

Wow! It has some color, and of course life. This is exactly what your IDE is purposed to do, provide you with an ideal environment to write your C++ code.What you see in color is what ’ s known as syntax highlighting, the segmentation of your program into various distinct parts.

There you are, having your development environment and first C++ program right before you. Can this get more exciting? Will see about that!

I will introduce you to variables, fundamental input and simple C++ mathematical concepts in the next chapter. We ’ ll explore them together and I ’ ll show you how to use them in your first C++ program, the one you just created.

Chapter 3

C++ Variables, Basic Input and Math Concepts

Here we take your adventure in learning C++ a notch higher. Have fun!

Data storage and retrieval are critical when it comes to computing.Programs and applications aren ’ t just developed to manipulate data, but also use them. Computer memory is where data storage takes place.

If you want to retrieve data, you must know the memory location where it ’ s stored. With variables, you can easily access stored data. They act as placeholders.

Variables are data stored in applications and programs. You can easily change or manipulate them. In C++, if you need storage for values, variables come in handy. In this language, you can interpret variables (data) in many ways.

Compilers allocate memory locations based on data types. Variables are these locations allocated in computer memory. Every variable declaration has a unique function and memory requirements.

Therefore, variables are names used to refer to memory locations as references by data value identifiers that you can easily change or manipulate in an app or program. Data types are the many ways in which data is stored in memory.

Variable Data Types

Here are the five major data types in

C++:

Char – this data type is used to store variable ASCII characters. These are usually enclosed in single quotes.

‘ c ’ ‘ x ’ ‘ a ’ ‘ * ’ etc

Float – this data type is used to store decimal numbers, also known as floating point numbers.These numbers comprise of integer and fractional parts with a decimal point separating the two. What ’ s more, they can either be negative or positive numbers.

.9 199955.68 0.6 10.2 etc

Int – this data type is used to store whole numbers or those with integral values. Although these can either have a negative or positive sign, use of commas aren ’ t allowed.

10 4 -35 22 -40 etc

Bool – this data type is used to store Boolean values which are either true or false.

Double – this data type is used to store double or large floating point values.

Variable Data Type Variations

The five data types also come in different variations for better control.

Short int – this is an integer that ranges from -45589 to 45589. Unlike normal integers, their short variations occupy less space.

Unsigned int – these are positive

integers, but double the range of normal positive integers.

Constants – constants are given values prior to being declaredand once they ’ re declared, they remain fixed.

Place *‘ const ’* before a data type to declare a constant despite the variable being declared.

C-string – although this is neither a data type nor is it declared, they showcase ‘ c-like ’ strings.They ’ re character arrays that can hold data with more than a single character.

Although the last two variable types aren ’ t critical now, they ’ re generally essential in other programming areas of application.

Practical Use/Example

Let ’ s now learn this by example.

Launch CodeBlocks and locate the ‘ File ’ menu.Under the new sub-menu that pops up, select ‘ New ’ then ‘ Project ’ .

Find ‘ Console Application ’ within the options available in the new window that opens up. Using the default options in the compiler, create your C++ practical project and type the name ‘ Variables ’ . Do this for each chapter, naming the project the respective chapter title.

On the active compiler window ’ s side bar, locate ‘ Expand Variables ’ . Click on it then select the ‘ Expand

Sources ’ option. Under the new lists, double click on ‘ main.cpp ’ .

Once the file opens, delete all the items that appear. Begin your practice by typing the following code in the file:

```
#include <iostream>

Using namespace std;

int main()

{

        int apartments = 5; // declare
variable “apartments”, give it a value
```

```
        cout << “I have” << apartments <<
“ apartments!\n “; //print

        return o;

}
```

Next, you need to compile your code then run it.

Your console should read: “I have 5 apartments!”

That’s right.

You did so well. Keep it up.

Comments

The double slash is used to denote comments, which come in two variations: single line and multi-line comments.

Single Line Comments – these can go anywhere within your program code. They tell the compile to ignore anything after that code line. For instance,

```
//this is your comment
```

Multi-line Comment – these start with a /* and end with */. Whereas /* indicates the beginning of the comment to the compile, */ terminates or ends it. Without indicating the end of your comments with */, the compiler treats everything that come after /* as comments. For instance,

```
/* this is
        your multi-line
comment */
```

Based on our comments in the code above, it ’ s clear that the variable we declared is known as ‘ apartments ’ . It was of the data type integer and we gave it a value 5.

Variable Manipulation

Next, you need to learn and practice how to manipulate variables. For instance, change the value of the integer from 5 to any other number, let ’ s say 10.

You ’ ll notice the program will display “ I have 10 apartments! ” after compiling and running the changed code.

You can also ask your users the number of apartments they own. Here ’ s where you ’ ll have more fun!

To do this, first delete everything on the file. Don ’ t worry about it because you ’ re not just learning more, but also having even much more fun.

Type the code below in the file:

```
#include <iostream>


Using namespace std;
```

```
int main()

{

        int apartments;

        cout << " How many apartments
do you have?\n";
          cin >>" << apartments;
          cout << "Oh My God! You have
" << apartments << " apartments!";

          return 0;
}
```

You’re probably wondering, “What’s going on here?” Don’t worry, just read on to understand this new code.

We have declared apartments as a variable, but in this case, no value was assigned to it. Although we can assign a value as in the previous code example, we seek a different result here hence we don’t need variable values in this case.

Since our aim is to ask our user(s) the number of apartments they own, we have used *cout* in the code to indicate ‘console out’ because it’s a standard method for outputs. We achieved this by adding a print out line to our code.

We need to enable users to input data for them to respond to the question asked. This is where the use of cin and (>>) come in. Whereas the former stands for ‘console in’, the latter is placed before the variable we expect from the user and acts as a stream operator of input.

Once we receive the user data or information, it’s printed back on the user console. Whatever the user enters or types and goes into the output stream is written to the console by *cout*. This is indicated by the (<<) stream output operator added to our code. Everything within quotes is a text string to be printed on the console just as it is.

Therefore, we use *cout* to imply an output stream. This means our program variables and/or strings can be printed out if everything is linked to each other.

What if you're interested in more things other than just apartments owned. Let's say you need to find more information about detached houses, condos, bungalows, etc.

If you need to ask a user(s) the number of apartments, detached houses, bungalows or even condos they won, here's what to do in C++:

```
#include <iostream>

Using namespace std;

int main()
```

```
{
        int apartments, detached houses,
bungalows, condos;
        cout << “ How many apartments
do you have?\n ” ;
        cin >> apartments;
        cout << “ How many detached
houses do you have?\n ” ;
        cin >> detached houses;
        cout << “ How many bungalows
do you have?\n ” ;
        cin >> bungalows;
        cout << “ What about condos?
\n ” ;
        cin >> condos;
        cout  << " Oh My God! You
have " << apartments << " apartments, "
<< detached houses << " bungalows,
and " << condos << " condos!\n " ;
                    return 0;
}
```

Similarly, just as we ' ve used the output stream to link various variables to each other, we can also use it to retrieve a stream of data.

For instance, check out the practical example below:

```
#include <iostream>
```

```
Using namespace std;

int main()

{
          int apartments, detached houses,
bungalows, condos;

          cout << " Enter the number of
apartments you own, followed by
detached houses, bungalows and finally
condos.\n " ;
          cin >> apartments >> detached
houses >> bungalows >> condos;
          cout << " Oh My God! You
have ” << apartments
<< “ apartments, “ << detached houses
<< “ bungalows, and “ << condos
<< “ condos!\n ” ;

        return 0;

}
```

As you can see, C++ is not just easy to use, but also comprehensible.

It ’ s also important that you know about enumerators. They might come in handy as you continue to learn this programming language.

Enumerators

Also known as iterators, enumerators enable programmers (you) to come up with ‘ state ’ statements. Although they ’ re simply integer values, you can easily utilize them to make comparisons. I ’ ll show you how to use enumerators in the next chapter.

Check below how you can declare enumerators:

Enum houses { apartments, detached houses, bungalows, condos };

Chapter 4

Conditions and Loops

I promise you that this chapter is even more interesting. You ’ ll definitely enjoy it!

Control Flow

Statements are executed from the first to the last. Executions begin from the first statement, followed by the second statement, then 3rd and so forth. This goes on until the last statement is executed. At this point, the program terminates.

If your program runs or executes similar statements sequentially, it would actually be useless. Depending on prevailing circumstances and what you want to do, your program should enable you to make necessary statement changes or select the ones to be executed.

A good example is when your program is designed to count the number of a specific word in a document. Irrespective of the document you want checked, your program should easily check any given file for any specific word, giving the correct number of that particular word.

This is what ’ s known as control flow. You ’ re able to change how your program executes statements. If you want your program to behave in accordance to your input, there should be a way for you to examine the latter. This is where conditions come in.

With a condition, your program checks

variable values, and depending on specific statements known as conditions, execution either takes place or not. The two main conditional structures used in C++ are:

If and switch case

Before we discuss these two conditional structures and I show you how they ’ re used practically, let ’ s have a look at operators.

Operators

Operators can either be logical or relational. With these operators, a compiler can determine if a condition in a statement is either true or false.

Relational Operators – these help

ascertain the relationship between any two expressions. Examples of relational operators are as follows:

>operator means **greater than**

>=operator means **equal to or greater than**

<operator means **less than**

<=operator means **equal to or less than**

==operator means **equal to**

!=operator means **not equal to**

Although relational operators work just like arithmetic operators, the former examines the relationship between two expressions, returning a Boolean value to indicate whether it ’ s true or false.

Note that a Boolean expression is one that returns the true or false values.

Logical Operators – logical operators, on the other hand, combine simple relational expressions to come up with complex Boolean expressions. Here are examples of logical operators you need to know:

&&operator means ‘ **and** ’

‖ operator means ‘ **or** ’

!operator means ‘ **not** ’

Here ’ s an example of the logic rules returning true or false:

Apartments	Condos	Apartments

		&& Condos
True	True	True
True	False	False
False	True	False
False	False	False

Apartments	Condos	Apartments || Condos
True	True	True
True	False	True
False	True	True
False	False	False

Apartments	!Apartments
True	False

False	True

The urinary operator (!) negates the value of a single argument.

Here ’ s a simple practical example: (Apartments = 10 and Condos = 4)

! (Apartments > 6) → False

(Apartments > Condos) && (Condos > 0) → True

(Apartments < Condos) && (Condos > 0) → False

(Apartments < Condos) || (Condos > 0) → True

In Boolean expression, any statement

with a value of 0 is false whereas that with any other value other than 0 is true.

If, If-else and Else-if

Thee **‘ if ’ condition**is an expression with a value that ’ s being examined. Your statements get executed if the condition is trueprior to your program continuing to run. However, if the condition isn ’ t true, the compiler ignores the statements.

You can write the ‘ if ’ condition in the following format:

```
if (condition to be met)
{
        Statement 1
        Statement 2
        …
}
```

In case of a single statement, you don ’ t have to use the curly braces. Check out the example below:

```
if (condition to be met)
        Statement
```

The **if-else condition** comes in handy when you need to examine two sequential statements in a block. In case the ‘ if ’ condition is ascertained to be true, the block statements corresponding to it gets executed. Conversely, if the condition is false, the ‘ else ’ statement is executed.

In this scenario, only a single statement block gets executed because the condition can either be met or not. The if-else condition is written in the following format:

```
if (condition)
{
        Statement 1a
        Statement 1b
        …
}
else
{
        Statement 2a
        Statement 2b
        …
}
```

However, omit the curly braces if your code has only a single block statement as in the following example:

```
if (condition)
     Statement 1a
Else
      Statement 2a
```

When your code has many conditions to be met with at least two block statements to be executed, the **else-if** condition comes in handy. A block statement in correspondence with the first condition is executed if the latter is met.

However, if the first condition isn ’ t met, only the block statement in correspondence to the ‘ else-if ’ condition is executed when the second condition is met. Here ’ s the format of the else-if condition:

```
if (condition1)
{
    Statement1a
   Statement1b
   …
}
else if (condition2)
{
   Statement2a
```

```
    Statement2b
    …
}
```

It ’ s possible for your C++ code to have at least one *else if* condition, each with a condition to be met. The moment a block statement is executed after its corresponding condition is met, the compiler ignores any other else if conditions.

This means that if your code has an if-else-if condition, either no block statement or one is executed.Though rarely, a C++ code can also have an if-else-if-else condition. In this scenario, the last else condition ’ s corresponding block statement is executed if none of the previous conditions are met.

Just like in a normal if-else condition, at least one block of corresponding statement must be executed.

Enough of the talking and theory, let ’ s do one practically. Go back to your CodeBlocks compiler. Delete everything in the file and type the code below:

```
#include <iostream>

Using namespace std;

int main()

{
        int Apartments = 6;
        int Condos = 3;
        if (Apartments > Condos)
        cout << " Apartments are more
than Condos\n " ;
        else if (Condos > Apartments)
        cout << " Condos are more than
Apartments\n " ;
        else
        cout << " Apartments and Condos
are equal in number\n " ;
        return 0;
}
```

Compile and run this program; see what

happens. The code returns *Apartments are more than Condos*.

If you interchange the values of Apartments with that of Condos (apartments=3 and Condos=6), the program will output *Condos are more than Apartments*.If you assign each variable a similar value, let ’ s say Apartments =6 and Condos =6, the program outputs *Apartments and Condos are equal*.

Switch-Case Condition

Just like the ‘ if, if-else and else-if ’ conditions we ’ ve discussed above, switch-case conditions can either execute certain statements when a corresponding condition is met or not.

However, the difference between these two types of conditions lies in the unique behavior and syntax of switch-cases.

```
switch (expression)
{
        case constant1:
        statement1a
        statement1b
        …
        break;
        case constant2:
        statement2a
        statement2b
        …
        break;
        …
        default:
        statement3a
        statement3b
        …
}
```

In our program code above, *switch* tests *expression*. All the statements that come after *case constant1* get executed, if *constant1* and *expression* are equal, until the compiler finds a *break*.

However, *expres*sion is compared to *constant2* if it ' s not equal to *constant1*. All the *statements* after *case constant2* are executed, if *case constant2* and *expression* are equal, until the compiler finds a *break*.

Otherwise, the process repeats for all the constants, one after another. If no match is found, all the statements that come after *default* get executed.

Note that in this instance, curly braces are important when it comes to enclosing the whole *switch-case*, but not when there ’ s at least a single statement; we attribute this to the distinct *switch-case* behavior.

Although you can use *if-else* condition equivalents in a switch-case, you can express the behavior in a better way.

It ’ s time for practice so let ’ s get back to our compiler.

Delete anything that ’ s on the file and type the following code:

```
#include <iostream>

Using namespace std;

int main()

{
        int Apartments=10;
        switch (Apartments)
{
        case 1:
        cout << “Apartments are 1\n”;
        break;
        case2:
        case3:
        cout << "Apartments are 2 or 3";
        break;
        default;
        cout << "Apartments are not 1, 2,
or 3";
        }
        return 0;
}
```

Compile the code and run it. See what happens.

This program will output *Apartments are 1, 2, or 3*. Replace *Apartments=10* with *Apartments=2* and note the new program output.

It will print out *Apartments are 2 or 3*.

Loops

Loops can help you repeatedly execute or run several statements until a certain condition is met. The given statements are run sequentially until a specified condition is declared as true.

The sequence of statements is held within a loop body, curly braces. Each time the loop body is executed, the compiler checks whether the statement is true or false. If it ’ s false, the loop body is executed again. This continues until the condition is ascertained to be true.

Types of Loops

Loops are available in three forms: *while*, *do-while* and *for*.

while and do-while

The ***while*** *loop* – thisloop is the same as the ‘ *if* ’ condition we discussed above. The code is written as follows:

```
while (condition)
{
        statement1
        statement2
        …
}
```

Statements in a block are executed one after another, repeatedly, until a condition is met. You don't have to use curly braces if there's just a single statement in your code.

It's clear now that every time we practice with a new code, you need to clear the compiler file to create room for our new program. I'm sure by now you've mastered this and find yourself automatically removing the code we worked on previously.

Well, let's move on.

Type in the following code:

```
#include <iostream>
```

```
Using namespace std;

int main()

{
        int Apartments=0;
        while (Apartments are less than
10)
        Apartments=Apartments + 1;
        cout << "Apartments are" <<
Apartments << "\n";
        return 0;
}
```

What's your program output after compiling and running this code? It should read something like:

Apartments are 10.

The ***do-while*** loop – this is a loop variation that ensures block statements are executed at least one time. You can code this loop as follows:

```
do
{
        Statement1
          Statement2
          …
}
while (condition);
```

If the condition is met, the block statements get executed, after which the program goes back to the beginning of the block. In this instance, you must use curly braces.

Note that the program ends with a semicolon just next to the *while* condition to be met.

for

for loop – this is similar to the *while* loop except for its different syntax. Check out the following code:

```
for (initialization; condition; increment)
{
        statement1
        statement2
        …
}
```

This loop enables an initialized counter variable at the point where the loop starts with an increase in every loop iteration. If your program has a single statement, consider not using curly braces.

We have to type a new code into the compiler file to remove everything on there.

```
#include <iostream>


Using namespace std;


int main()
```

```
{
        for (int Apartments = 0;
Apartments < 10; Apartments =
Apartments +1)
        cout << Apartments << "\n";
        return 0;
}
```

Compile and run the code. What's the output?

The numbers 0 to 9 will be printed out, each value on its own line.

If you've pre-defined the counter variable, there's no need for a new definition in the *for* loop's initialization part. The following program code, therefore, is valid:

Clear your compiler file and type the following:

```
#include <iostream>


Using namespace std;


int main()


{
        int Apartments = 0;
        for (; Apartments < 10;
Apartments = Apartments + 1)
        cout << Apartments << “\n”;
        return 0;
}
```

The first semicolon instance within the *for* loop’s parentheses is necessary. You can express a 'for' loop as a while loop, and the latter as the former. With the *for* loop structure or form we’ve already written above, let’s write a *while* loop that’s equivalent as follows:

```
Initialization
while (condition)
{
        Statement1
        Statement2
        …
        Increment
}
```

Based on our *for* loop example above, we can write a *while* loop as follows:

```
#include <iostream>

Using namespace std;

int main()

{
        int Apartments = 0;
        while (Apartments < 10)
{
            cout << Apartments << “\n”;
            Apartments = Apartments + 1;
}
        return 0;
}
```

Although you can place the increment step within the block statement, making it the last step is a good practice, especially if the former statements make use of the prevailing counter variable value.

Nested Conditions for Control Flow

Nested conditions are loops placed within loops and if conditions placed within if conditions. We accomplish this by either placing loops or if conditions within statements.

Let’s look at this if condition placed within another if condition to form a nested loop. Clear your compiler file and type the following:

```
#include <iostream>

Using namespace std;

int main()

{
        int Apartments = 10;
        int Condos = 0;
        if (Apartments > Condos)
{
        cout << “Apartments are more
than Condos\n”;
        if (Apartments == 10)
        cout << “Apartments are equal to
10\n”;
        else
        cout << “Apartments are not equal
to 10\n”;
}
        else
        cout << “Apartments are not more
than Condos\n”;
        return 0;
}
```

This program will send two outputs to the printing console. It will print out that:

Apartments are more than Condos and in the next line,

Apartments are equal to 10.

Now that you've learnt and can now write a nested condition where an if condition is placed within another, let me show you how to go about writing nested conditions with loops placed within loops.

Type the following code in the compiler file after clearing it:

```
#include <iostream>
```

```
Using namespace std;


int main()
{


        for (int Apartments = 0;
Apartments < 6; Apartments =
Apartments + 1)
{
        for (int Condos = 0; Condos < 6;
Condos = Condos + 1)
        cout << Condos;
        cout << “ \n ” ;
}
        return 0;
}
```

Compile the code and run it. The output stream will display 0123 in four lines, sequentially.

Note that you can easily exit a loop by writing a *break* statement.

Wow! How easy was that?

You ’ ve now learnt about loops, variables and conditional statements. I believe you can now confidently write a simple beginner-level program with the C++ concepts you ’ ve learnt so far.

You ’ re now half-way the journey of becoming a programmer in C++. In the next chapter, we ’ ll discuss C++ functions to take the knowledge you ’ ve gained by now a notch higher.

Chapter 5

C++ Functions

Functions in C++ are simply sub-programs written to perform specific pre-defined roles. Different parts of your program work by calling other program parts. You must define your program function at least once and call up other program parts.

3 Important Parts of a Function

Your typical function should have arguments, a name and even a body. These are discussed below:

- **A Function Name** – this is the name you use when referring to a function. The name of a function, unlike that of an object, is a verb.

- **Return Type** – functions play the role of returning variable types. It can either be a string or an integer.
- **Parameters** – these are elements that can be taken by a function. Moreover, you can easily alter parameters using functions.

Taking and Returning Integers

Functions can take in an integer Apartments and return an integer as follows:

```
int return_int (int Apartments)
{
        return Apartments;
}
```

In the code above, *int* is our return type whereas *return_int* is the name of our function. Integer Apartments is our parameter. In simple terms, this function takes in the integer *Apartments* and returns integer *Apartments*.

Calling a Function in Main

We ' ll call our function above in main. Let ' s get started.

Create an integer *Studio*. We ' ll have it as our *return-int* output. Our function will take in a 6.Let ' s now write this code; clear your compiler file and type the following code:

```
#include <iostream>
int return_int (int Apartments)
{
        return Apartments;
}
int main ()
{
                int Studio = return_int (6);
                std:cout<< “ Hello World,
I ’ m Here Now!\n ” ;
}
```

Before we can test our code, let ’ s replace “ Hello World, I ’ m Here Now!\n ” with Studio. Your new code should look like the one below:

```
#include <iostream>
int return_int (int Apartments)
{
        return Apartments;
}
int main ()
{
                Int Studio = return_int (6);
                std:cout << Studio;
}
```

What appears in youroutput console? It should be ‘ 6 ’ .

A Function that takes in Integers

Apartments and Condos and Returns their Sum, which is also an Integer

Type the code below in the compiler file after clearing everything in it:

```
#include <iostream>
int add_ints (int Apartments, int Condos)
{
        int Houses = Apartments +
Condos;
        return Houses;
}
int main ()
{
                Int Homes = add_ints (6, 5);
                std:cout << Homes;
}
```

Our function will add 6 to 5 and store the output 11 in the *Homes* variables for printing.

Compile and run it. It should output 11.

String Functions

Let ’ s write a code that takes in two strings and adds them. We ’ ll use the function *combine_strings* to add strings Apartments and Condos. You ’ ll notice some similarity with our previous code; we just change a few things here and there.

Type code below in a cleared compiler file:

```
#include <iostream>
std: string combine_strings (std:: string
Apartments, std:: string Condos)
{
                std:: string combined =
Apartments + Condos;
                return combined;
}
int main ()
{
                Std:: string Houses =
combine_strings( “ Apartments ” , “ Con
                std:cout << Houses;
}
```

In this code, our strings *Apartments* and *Condos* will be added; this is similar to adding integers as we previously did. The two strings will be combined and the result printed out; this is the role of the *return combined* function.

In computing, this combination or addition is what's known as *concatenation*.

Now compile and run the program. What's displayed?

Apartments, Condos appear in that order.

Functions and if Statements

Let's create a function *more* and have it take in two integers Apartments and Condos, and return an integer.

```
#include <iostream>
int more (int Apartments, int Condos)
{
        if (Apartments > Condos)
{
        return Apartments;
}
else
{
        return Condos;
}
}
```

Our *more* function takes the two integers *Apartments* and *Condos* in its parameters and compares them. It only returns the integer that's more than the other. For instance, if Apartments are more than Condos, it returns the latter. The converse also applies.

Let's call our main function integer before we can run our code in the compiler. We'll create an integer *Houses* to hold the outcome of the *more* function. It'll be like:

```
          int Houses = more (12, 6);
```

Our final code will look like the following (clear the compiler file and type it because we now need to test it):

```
#include <iostream>
int more (int Apartments, int Condos)
{
        if (Apartments > Condos)
{
        return Apartments;
}
else
{
        return Condos;
}
        int main ()
{
        int Houses = more (12, 6);
        std:cout << Houses;
}
```

Compile and run the code. The two integers are compared and your program will display 12 as the output.

Void Functions

Although we've just explored string and integer functions, a function can also take in and return doubles, floats, chars or even bools. Before we finalize this chapter, let me show you how to write a void function.

Void functions return no value. We use this when we expect our code or program to return no value.

Let's practice: type the following code in the compiler file after clearing it.

```
#include <iostream>
Void SquareArea ()
{
        double Length;

        cout << “\nType the length of the
square:“;
        cin >> Length;

        cout << “\nSquare attributes:”;
        cout << “\nLength = “ << Length;
        cout << “\nArea = “ << Length *
Length;
}
```
You can’t display a *void* function on a similar line to the *cout* output extractor.

What’s more, you can only call a *void* function, but can’t assign it a variable.

Compile and run our *void* function. As you can see, nothing is returned as I have explained above.

Congratulations! You’re now skilled in C++ functions and can actually code functions.

Before we continue learning other important aspects of C++, we’ll review a useful working code in the next chapter.

Chapter 6

Review of a Weight Converter

I am going to provide you a working code below for review and practice. Feel free to play around with it and later it as you deem fit to see what happens to the output when the changed code is compiled and run.

Review the Code Below

With all the explanations I ’ ve made thus far and the skills you possess now, the code is straightforward.

The code converts ounces and pounds to grams and kilograms. It enables you to input various values until decide to close the program. Enjoy!

```
#include <iostream>

using namespace std;

void convertWeight(double ounces,
double &grams, int pounds, int
&kilograms);
double output(double ounces, int
pounds);

int main()
{
   double ounces;
   int pounds;

   char answer;
   do
   {
      cout << "Enter weight in pounds
and ounces: " << endl;
      cin  >> pounds >>  ounces;
      cout << output(pounds, ounces) <<
endl;

      cout << "Do you want to test again
(y/n)?: " << endl;

      cin >> answer;

   }
   while (answer == 'y' || answer == 'Y');
   return 0;
}

double output(double ounces, int
pounds)
{
   double grams;
   int kilograms;

   convertWeight(ounces, grams, pounds,
kilograms);

   cout << pounds << "pounds and " <<
ounces << "ounces = "
       << kilograms << "kilograms and "
<< grams << " grams."  << endl;



   return 0;


}


void convertWeight(double ounces,
double &grams, int pounds, int
&kilograms)
{
   kilograms = (pounds +
ounces/16)/2.2046;
   grams = kilograms * 1000;
}
```
Note that this converter makes use of all

the concepts we ’ ve learnt. Type it in the compiler file and run it.

In the next chapters, I ’ ll introduce you to pointers, arrays and strings. Are you ready?

Let ’ s move on.

Chapter 7

Arrays, Pointers and Strings

Arrays

When you group similar items of the same nature, the group is known as an array. Arrays can either be single or multi dimensional.In this book, we ’ ll only cover the former.

Array Declaration

Arrays must be declared being utilized. We must also define the type of array in use for the compiler to know. This ensures that enough memory space is reserved for our array. Therefore, when we declare an array, each array member is placed in its reserved memory location.

Just like variables, declare an array as follows:

Datatype ArrayName[Order\Dimension]

Here ’ s an example:

int weight[100];

The square brackets define our array ’ s size or dimension. Note that arrays can be of different data types such as char, float, int, etc.

Array Initialization

We can initialize one-dimensional arrays as follows:

Datatype ArrayName[Order\Dimension] =

```
{item1, item2, …, itemn};
```

Type the following code in the compiler file:

```
#include  <iostream>
using namespace std;
int main()
{
        double weight[] = {34.66, 450.45,
23.58, 5.33, 54.04};
        cout << " 1stmember = " <<
weight[0] << endl;
        cout << " 3rdmember = " <<
weight[2] << endl;
```

```
        return 0;
}
```

Compile the code and run it. What's the ouput?

It should display:

```
1st member = 34.66
3rd member = 23.58
```

If you want to access all the items or array members, just do the same, listing the items from the first to the last in the array list.

Pointers

When you declare a variable, the computer reserves memory space for it, utilizing its name to refer to the space reserved in memory. In computing, this space is referred to using an address. This means that all declared variables have an address. The address of declared variables can be found.

When you find the address of your variable, you can use it. Every time we pass an argument (variable) to a function, just like a reference, its address is used. As a result, the calling function accesses the variable ’ s address to directly use its value. This means that we can change the value of a variable through the calling function.

In normal scenarios, the previous variable value is maintained once you exit the calling function. However, with pointers, a called function can return many values even if it was initially declared as void.

When arrays are declared, their dimensions must be defines.Unlike arrays, pointers can help us code without the need to know or find array dimensions. This is because they ’ re better memory managers.

Since arrays can comprise of whole texts and long characters, pointers come in handy due to their ability to hold arrays of different sizes. This means, we can use pointers instead of arrays, eliminating the need to know the size of the given array. This is taken care of by the compiler.

As in the case of arrays, functions can take in pointers and return altered values even the case of void functions.

Pointers are ideal when we need to work with multi-dimensional arrays. Instead of direct referral to the address of a variable, we can declare another variable to do the reference.

Therefore, pointers are variables that reference or refer to the address of other variables. Just like any other variable, pointers must be declared and initialized prior to utilizing it.

Declaring Pointers

We use an identifier (data types), an asterisk (*), pointer name and semi-colon (;) in that order, to declare a pointer.

The format is as follows:

```
Datatype * Pointername;
```

The data type or identifier you pick should be that of the variable whose address is to be referenced. This means that if the variable is an integer, the data type becomes an integer. The asterisk identifies the variable being declared as a pointer.

The asterisk must come before every variable to be declared as a pointer, each separated by a coma as follows:

Datatype* pointer1, *pointer2;

You can also declare them in different lines as follows:

Datatype* pointer1;

Dataype * pointer2;

It doesn ’ t matter how you append the asterisk, as long as it appears between the identifier and the variable to be declared as a pointer. In case you place the asterisk before each variable to be declared a pointer, only the first variable is considered a pointer and the rest normal variables.

Check this out:

```
        Datatype * pointer1, pointer2;
```

Pointer2 isn ’ t a pointer, but a normal variable.

Naming and Initializing Pointers

Since pointers are variables, they use the same naming system we used in a previous chapters when covered variables.

When we initialize a pointer, just like in the case of variables, we tell the compiler what it ’ ll do.

int pointer;* a pointer declared in this way is initialized as follows:

int pointer = &Variable;* (removing the asterisk in the initialization but keeping the & before the variable also works when you need to do the initialization in a different line.)

Let ’ s practice a bit: type the following code in the compiler file.

```
#include  <iostream>
using namespace std;
int main()
```

```
{
        int value = 22;
        int* pointer = &value;

        cout << " Value is at: " << Value
<< " \n " ;
        cout << " Pointer is at: " <<
*Pointer;
        cout << " \n\n " ;
        return 0;
}
```

Compile this code and run it. The program outputs the following results:

```
Value is at: 22
Pointer is at: 22
```

You can also initialize the pointer as

follows: the result will be the same.

```
#include  <iostream>
using namespace std;
int main()
{
        int value = 22;
        int* pointer;

        cout << “ Value is at: ” << Value
<< “ \n ” ;

        pointer = &value;
        cout << “ Pointer is at: ” <<
*Pointer;
```

```
        cout << “ \n\n ” ;
        return 0;
}
```

We can also declare the variables first then and do the initialization later as follows:

```
#include  <iostream>
using namespace std;
int main()
{
        int value;
        int* pointer;
        value = 22;
        pointer = &value;
        cout << " Value = " << Value
<< " \n " ;
        cout << " *Pointer = " << *Pointer
<< " \n " ;


        cout << " \n " ;
        return 0;
}
```

We can change the value of a variable that was initially declared and assigned a pointer by giving it a new value as follows:

```
#include <iostream>

using namespace std;



int main()

{

   int value;

   int *pointer;



   Pointer = &value;
   Value = 56;

   cout << "Value = " << value << "\n";

   cout << "*pointer = " << *pointer <<
"\n";



   Value = 15;

   cout << "Value = " << value << "\n";

   cout << "*pointer = " << *pointer <<
"\n";
```

```
    cout << "\n";

    return 0;

}
```

Let’s now type the following code into the compiler and see what happens.

```
#include <iostream>

using namespace std;


int main()
```

```
{

        int value;

        int *pointer;

        Pointer = &value;

        Value = 56;

        cout << "Value = " << value <<
"\n";

        cout << "*pointer = " << *pointer
<< "\n";
```

```
        Value = 15;

        cout << "Value = " << value <<
"\n";

        cout << "*pointer = " << *pointer
<< "\n";

        *pointer = 100;

        cout << "Value = " << value <<
"\n";

        cout << "*pointer = " << *pointer
<< "\n";
```

```
        cout << "\n";

        return 0;

}
```

Compile the code and run it. The output should read:

```
Value = 56
*Pointer = 56
Value = 15
*Pointer = 15
Value = 100
*Pointer 100
```

Strings

An array of characters ending with a terminating null character (\0) makes up a string. The std namespace string library is commonly used when you want to define a string.

There are many string functions you can use to manipulate strings. They include the following:

- *strlen()* function used to determine the length of a string.
- strcat() function used to append or add the character of two strings.
- strncat() function used to append or

add specific characters from the source string to those in the destination string.

- strcpy() function used to copy one string into another, replacing the latter.
- strncpy() function used to copy specific characters from a source string to a destination string, replacing the latter.
- strdup() function used to make a duplicate or copy of a string.
- strcmp() function used to compare a couple of strings, returning an integer as an outcome of the comparison.
- strncmp() function used to compare

specific characters of a string with those of another string. It returns an integer as an outcome of the comparison.

- stricmp() function used to compare two strings irrespective of their case, meaning it doesn ’ t matter whether capital and small letters are mixed or not.
- strincmp() function used tocompare two strings irrespective of their case, meaning it doesn ’ t matter whether capital and small letters are mixed or not. However, the comparison is based on a specific number of characters.
- sprintf() function used to format strings and defines the manner in

which they should be displayed.

- strupr() function used to change or convert a string in small letters to uppercase letters.
- strlwr() function used to change or convert a string in capital letters to lowercase letters.
- strstr() function used to search for the first sub-string occurrence within a different string. It returns the remaining string as a new set of characters or string.
- strrchr() function used to screen strings from the right side in search of a particular character.
- strchr() function used to search for a particular character ’ s first

occurrence in a string.

Since we ’ re just learning basics in C++ programming language, we won ’ t delve into all the functions. However, it ’ s to know them and what they do.Let ’ s practice with the second string function in the list above.

Type the following code into the compiler file:

```
#include <iostream>

using namespace std;

int main()
```

```
{
        char *Make = "Suzuki ";
        char *Model = "Wagon";
        cout << "Historically, Make = "
<< Make;
        strcat(Make, Model);
        cout << "\n\nAfter concatenating,
Make = " << Make << endl;
```

```
        return 0;


}
```

Compile this code and run it. What ’ s the output?

The output will read: Historically, Make = Suzuki

After concatenating, Make = Suzuki Wagon

You ’ ve now learnt the important basics in C++ arrays, pointers and strings to enable you write simple codes.

In the next chapter, we ’ re going to finalize this book with file I/O and classes in C++.

Chapter 8

File I/O and Classes in C++

File I/O

In this chapter, we ’ re going to learn about file I/O in text files.Note that this topic is more advanced than what I ’ ll cover here.

There are two basic classes in C++ responsible for dealing with files. These include *if* and *of* streams written as *ifstream* and *ofstream*, respectively. You must use the header file *fstream* when using these two classes in your program.

Whereas we can use ifstream to read from files or input data into files, ofstream facilitates writing to files or file output.

Declaring File I/O Classes

Declare either of the two classes, *ifstream* and *ofstream*, as follows:

ifstream a_file; or ofstream a_file;

or

ifstream a_file (“ nameoffile ”); or
ofstream a_file (“ nameoffile ”);

If you pass the file name as an argument or variable in both the classes, a file is actually opened. Both classes also use open and close commands.

(a_file.open()) is an open command whereas a_file.close()) is a close command used by both classes.

However, since programs automatically

call the close commandwhen they terminate, you don ’ t have to use it. However, it comes in handy when there ’ s a need to close a file before the program the program terminates.

What makes C++ appealing when it comes to file handling lies in the simple nature of the file I/O functions. Using << and >> before a class instance is possible because overloading operators are supported in C++. Actually, after file streams are opened, we can use them as cin and/or cout.

Clear the compiler file and type the following code:

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

```
using namespace std;

int main()
{
  char str[10];

  //Creates an ofstream instance, and
opens variables.txt
  ofstream a_file ( "variables.txt" );

  // Outputs to variables.txt via a_file

  a_file<<"This text will be inside
variables.txt";

  // Close the file stream completely
  a_file.close();
  //Opens for file reading

  ifstream b_file ( "variables.txt" );

  //Reads a single string in the file

  b_file>> str;

  //Outputs 'This'

  cout<< str <<"\n";

  cin.get();    // Nothing happens until a
key is pressed

  // b_file closes completely at this point
}
```

Compile and run this code to see what happens.

By creating an ofstream that ’ s not in existence, you activate file opening in default mode. However, if the close command construct exists, clear or delete everything. It ’ s also possible to declare a second variable to define how you want the compiler to handle the file. Consider the following:

File appending using *ios::app*

Setting the prevailing position at the end using *ios::ate*

Deleting all the items in a file using *ios::trunc*

You can append new data upon opening a file without tampering with its contents. Use the function below:

ofstream a_file (" variables.txt " , ios::app);

if you were not able to open a file, don ' t use the function to open it. Use the code below to test it first:

```
ifstream a_file ( "variables.txt" );

if ( !a_file.is_open() )

{

 // The file couldn't open

}

else
```

```
{
  // Use the file stream safely
}
```

Classes

With classes, you can create custom or tailored data types. The keyword class is used. Classes can either be empty or assigned values. Once created, classes are used just like normal built-in data types such as char, int, bool, etc.

For instance, let ’ s declare the class books in the code below:

```
#include <iostream>
```

```
using namespace std;

class Books
{
        int numberOfpages;
};

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

If you want to declare an empty class, remove the function below from the code above:

```
{

        int numberOfpages;

};
```

Public (accessible to functions within and outside a class), protected (accessible to functions within a class only) and private (similar to protected) are the three restriction levels in C++ classes. You can declare them using their respective keywords and ending them with a semi-colon.

Restrictions make classes easier to use. Constructors and destructors must be featured in a class. Whereas the former initializes variables, the latter cleans up memory to free space allocated to variables.

Compilers also create default constructors, if you don ’ t need them in your program for initializations. The same applies to destructors. This means that constructors are called automatically whenever you declare a class instance.

When a scope is reached, your program terminatesor allocated memory space deleted, the destructor is automatically called. I know some of these concepts might sound like jargon (e.g scope), but don ’ t worry. You ’ ll learn them as you climb the C++ knowledge ladder.

What ’ s important now is that you understand the concept of constructors and destructors. Note that both of them don ’ t return variables or arguments so you wouldn ’ t want to use them for that purpose.

Give destructors and constructors a public restriction to enable you create classes. This is attributed to the fact that every time a class is created, the constructor is called. Doing otherwise brings about compiling errors.

Our lesson ends here!

I wish to congratulate you for having completed studying this book. Although the book ends here, learning doesn ’ t end here. Go through the book as many times as you can and have fun each time!

Conclusion

Thank you again for downloading this book!

I hope this book was able to help you to learn important yet basic C++ programming skills you need as a beginner developer. Note that every concept discussed in each chapter is basic knowledge you need to know as you embark on your journey to becoming a seasoned C++ programmer.

The next step is to practice, practice and practice. Use the codes in this book to practice and even try to write your own. I also recommend that you try using different compilers as you practice. This way, you ’ ll find one that you like for consistent future use. When you practice, you don ’ t just understand the concepts explained better, but also master the skills. I wish you all the best in your newly found career in programming. Hope you enjoy it every step of the way!

Finally, if you enjoyed this book, then I ’ d like to ask you for a favor, would you be kind enough to leave a review for this book on Amazon? It ’ d be greatly appreciated!

Thank you and good luck!