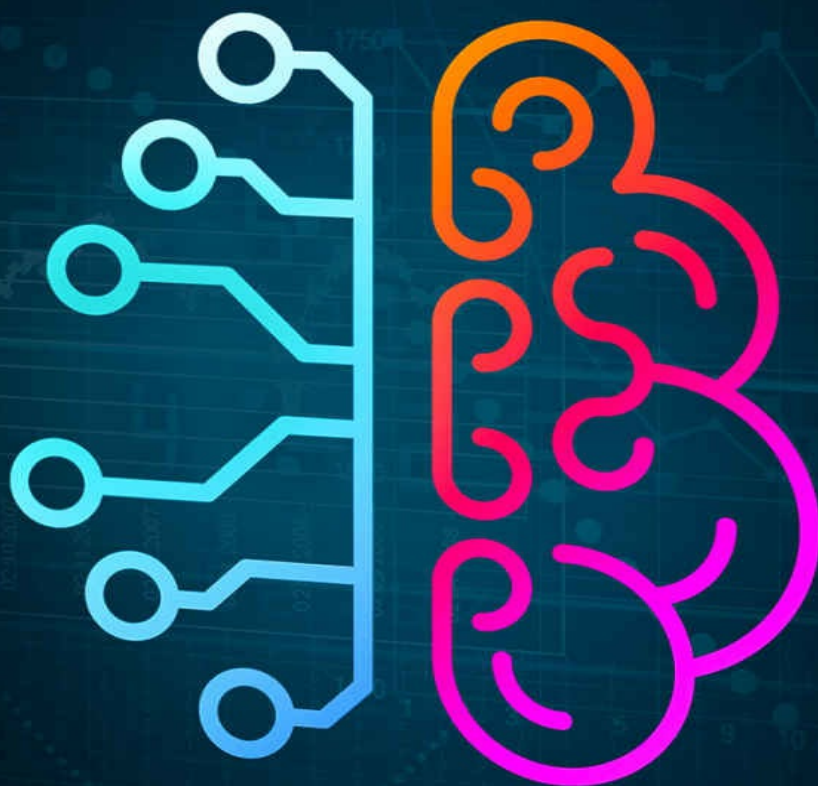


C++

THE ULTIMATE BEGINNERS GUIDE TO LEARN
C++ PROGRAMMING STEP-BY-STEP



M A R K R E E D

C++ Programming

*The Ultimate Beginners Guide to learn C++
Programming Step-by-Step*

Mark Reed

© Copyright 2020 - All rights reserved.

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

Legal Notice:

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaged in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

Table of Contents

Introduction

Chapter 1: Setting up a C++ Development Environment

Setting up Your Text Editor

Windows IDE Installation and Setup

Mac OS IDE Installation and Setup

Linux Compiler Installation

Chapter 2: Basics of C++, Principles of Programming

Principles of Programming

Overview of the C++ Syntax

First Program: Output and Basic Strings

Chapter 3: Variables and Data Types

Input and Output: Declaring Variables

Basic Data Types

Power of C++: Advanced Strings

Chapter 4: Operations in C++

Binary Operators

Unary Operations

Ternary Operators

Chapter 5: Decision Making in C++

Loops

If()...Else

Switch

Chapter 6: Creating Functions

Create and Call a Function

Parameter Passing

Function Overloading

Conclusion and Final Notes

Expanding Your Practice: Preparing Your Coding Environment

Coding Best Practices: Ownership of Learning and Collaboration

Take Away: Computer Science Concepts in C++

Glossary

Index

References

Introduction

This book is fit for beginners and for coders who are interested in getting into backend programming. Although C++ is sometimes portrayed as a specter of days past, the language is still with us and it continues to be behind some of the biggest technologies we use today—not mentioning its big imprint in the gaming world. Its power and versatility continue to make it one of the most important languages of our time. It is not going anywhere and learning it will expand your horizons.

C++ is often utilized as a backend language for big data because of its little processing overhead. Companies like Spotify, Adobe, YouTube, and Amazon power their backend with C++, and you will soon see why.

C++ is also behind powerful gaming engines. Gaming engines allow programmers to build a game without coding everything from scratch and to effectively render content. The Unity Game Engine and the Unreal Engine are examples of gaming engines that run on C++.

C++ is a beautiful, efficient language because of the favorable power/hardware ratio: it uses little hardware for the amount of power it gives us. This is why those who learn it love it.

In this book we will cover the following topics:

- Programming terminology and principles in programming
- Setting up a C++ environment
- Getting Started: Syntax, Data Types, and Variables
- Power of C++: Operations, Loops, Switches, and Decision Making
- Creating custom functions in C++

You will also find a useful glossary at the end so that you can use the book as a reference once you get cracking.

Chapter 1:

Setting up a C++ Development Environment

At its most basic, programming is writing a list of instructions in code that the machine can understand. The code resides in executable files. These files come with file extensions that tell a compiler what language is in the file. These extensions are the suffixes you often see at the end of the file, like “.js”. “.cpp” or “.hpp”.

To write code and save it in an executable file you need the following things:

1. A **text editor**: this will allow you to write and edit the code.
2. A **language compiler**: This program takes the code you have written and translates it into machine language that your computer can understand and follow.

All programming languages work like this except HTML, CSS, and JavaScript - these programs are interpreted and executed by the browser (“Introduction,” n.d.). This means browser languages like JavaScript are software-based, while C++ is compiled and then run directly on your machine, not in a software environment.

This means C++ is an assembly language. **Assembly languages** are low-level programming languages that need a compiler so they can run on a machine (Lithmee, 2018). In this context, the word low-level does not carry a bad connotation; it is descriptive, meaning that the language is closer to the machine or just a step away from it.

As you can probably guess, C++ is a general-purpose language that can run almost anywhere. This means it can be assembled and compiled in several different ways. This will largely depend on your operating system and the creation utilities you are using.

Our C++ exercises will be compiled on an online IDE. IDE stands for **Integrated Development Environment** and it is used to edit and compile

code. I bet that description sounds familiar. Yes, an IDE is an example of a text editor, but unlike a plain text editor, it has extra features that are important to the programming process. An IDE can do things like compile code, debug code, highlight code, warn you of syntax errors, and more.



Geeks for Geeks IDE: A web-based Program Compiler

This IDE can be found at <https://ide.geeksforgeeks.org/> and has several programming languages. We will be focusing on C++. This IDE has many coding utilities: a tabbed working space, an input box, and a code manager. *This code manager includes the two bottom buttons highlighted on the left.* These code manager buttons allow you to download and upload code as files with their corresponding file extension. This IDE also allows you to run and generate a URL that saves the result. We'll be using these generated URLs to manage our lessons.

In this book, we will use a Geeks for Geeks web-based IDE, but you should learn to set up a local IDE. For the majority of your programming career, that is where you will be working. Plus, you can customize the IDE to fit your needs and spruce up your code.

Setting up Your Text Editor

IDE environments that are focused on programming always have to have a text editor and a compiler within them. Non-IDE environments separate compilers and plain text editors. The text editor serves as a programming interface in non-IDE setups; this simply means the text editor will be the place you tinker with the code.

Note: If you have a Linux, you will already have a text editor as Linux is a console-based environment. Console-based environments use a **command-line interface (CLI)**, where you interact with programs by issuing lines of code. IDE for Linux may include a **graphical user interface (GUI)**. A GUI allows you to interact with programs through visual indicators such as clicking icons.

When looking for a text editor, you need one with syntax highlighting and indenting as all programming languages follow their syntax. This is because you want to be able to read your code easily and you want collaborators to be able to do so, too. These text editors help by improving readability. This is especially important because coding is no longer and has never been a solitary task. There is no one-man genius like in the movies.

Github and Pastebin are code aggregators that have syntax highlighting add-ons enabled. Github will allow you to host your entire project on their site, while Pastebin only allows code snippets. On these platforms, you can save code in a variety of languages.

They are very useful to programmers because they allow programmers to share code, collaborate, test, and so forth. Learning how to deploy a project to Github is one of the most important things in programming because it has become so standardized. So, maintaining a Github profile has also become important, as it holds all the projects you are working on, have worked on, and your activity (Peshev, 2017). To a potential employer or collaborator, this information is invaluable.



```

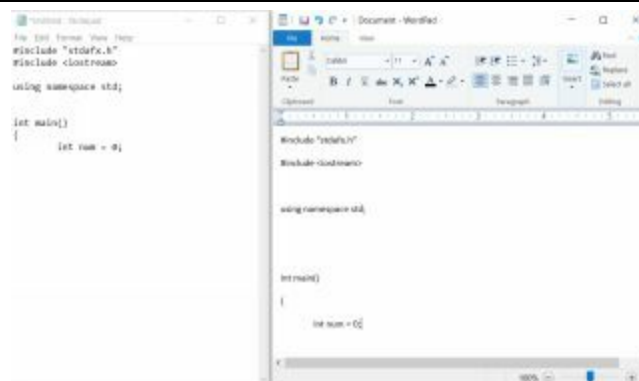
1 #include "stdafx.h"
2 #include <iostream>
3
4 using namespace std;
5
6
7 int main()
8 {
9     int num = 0;
10    int result = 0;
11    cin >> num;
12    for (int i = 2; i <= num; i += 2) {
13        result += i;
14    }
15    cout << endl << result << endl;
16    system("pause");
17    return 0;
18 }

```

C++ Code Syntax Example

This is a screenshot from Paste Bin with syntax highlighting enabled ([“C++ Code,” 2015](#)). Syntax highlighting and tabbed spacing help with making the code more comprehensible. All text editors with syntax highlighting will use this scheme: libraries in green, and functions, data types, and data in blue. Strings will show up as red.

When you are working locally you will not have luxuries like these readily available. The first text editor you will find on your system if you are using windows is Notepad. Wordpad is another one that has more GUI features. What you will notice as you open these programs is how plain and boring they are. They are like Word but worse, because they shouldn’t be simple word processors if writing efficient, elegant code is important to us. Their word processor-like aspects make them more suitable for writing words in them, not code, although you can code in them.



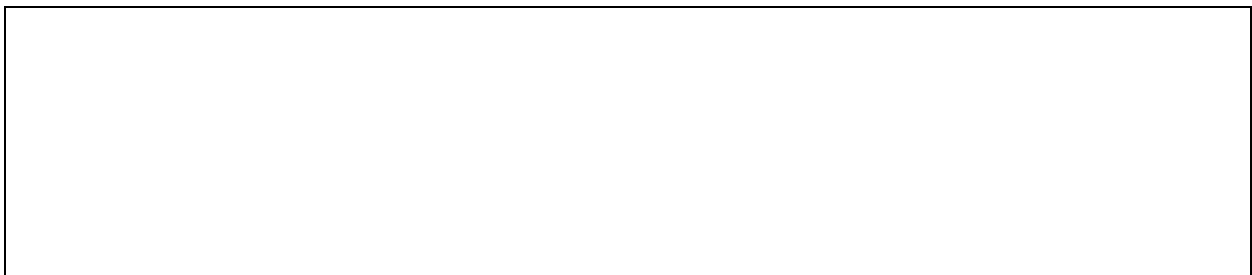
Side-by-side comparison of Windows' Notepad utility and Wordpad utility

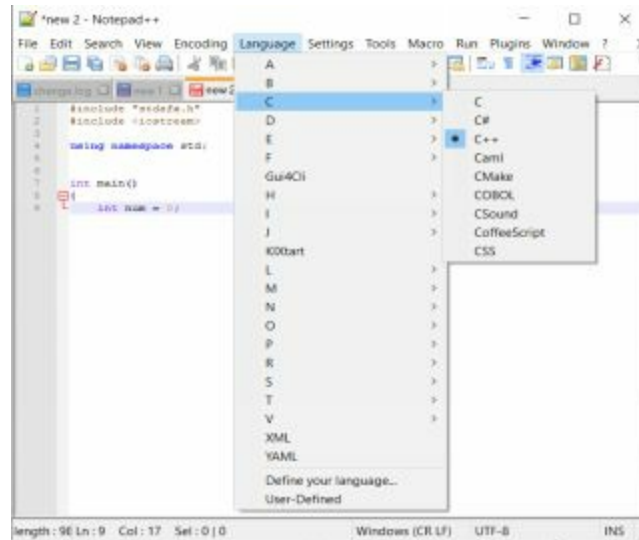
This is a screenshot that features two of Windows' built-in utilities, Notepad on the left and Wordpad on the right. Wordpad has more Microsoft suite GUI features that may be more recognizable to you, including Microsoft's quick access bar located above the highlighted "File" tab. Notepad is very bare-bones in comparison. Both utilities, unlike word processors, allow you to save in different file extensions. However, Wordpad is more similar to a word processor than Notepad, saving files in rich text format (.rtf). Wordpad will warn you that you will lose formatting if you save in an extension other than rich text format.

What makes Notepad and Wordpad unique is that they can save files in a variety of file extensions, while word processors cannot. File extensions are important because they tell the compiler how to interpret what is written in the file, and that leads to the machine having a set of instructions it can understand and execute.

Despite their abilities, these two programs lack crucial text editor features like syntax highlighting. You might think this is no big deal, but it is; the same words in code can mean different things because of how and when they appear, so highlighting helps us distinguish what they refer to. So if everything is plain, black and white, you have to work harder to figure out what a piece of code refers to. It sounds complex now, but once you code this will become obvious and necessary.

Notepad++, not to be confused with Windows' Notepad, has highlighting features but you will have to activate them like so:





Screenshot of Notepad ++ with C++ Syntax

Notepad ++ is free and, as the name suggests, was programmed in C++. This screenshot illustrates how to enable language syntax. C++ syntax can specifically be enabled by going to language >"C" > and navigating to C++. Notepad++ is only available on the Windows platform (Orin, n.d.).

Bluefish is a more advanced text editor that comes with more features ("Bluefish Editor: Features," n.d.). Unlike Notepad++, Bluefish is available on multiple platforms other than Windows. But I would not recommend Bluefish for beginners because it has a lot of features that can be overwhelming to a complete beginner. If you are not new to programming you can go give it a shot; you will find it has many of the GUI features typical of an IDE text editor.

Once in the text editor, you must select a compiler. Now let me show you how you would set up your local environment on three platforms: Windows, Linux, and Mac.

Windows IDE Installation and Setup

All you need to do is to install an IDE. Just remember that it has to have a text editor and a compiler within it. **Code::Blocks** is a useful, open-source IDE made for C++ and it easily fits with a variety of compilers including Microsoft's Visual C++. Installing Code::Block is easy. You just have to go to their downloads page at <http://www.codeblocks.org/downloads/26> and select the latest version.



Code::Blocks Download Page

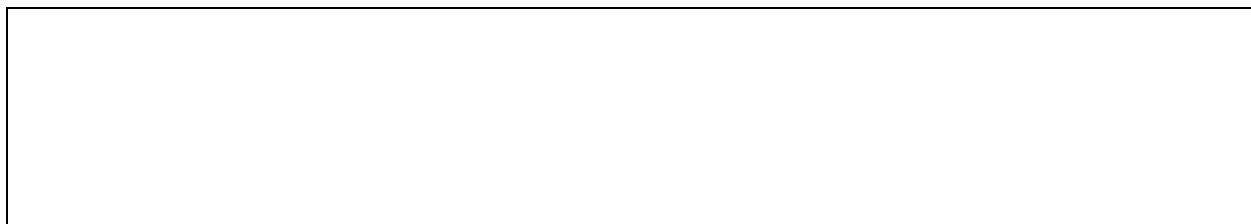
As of this writing, Code::Blocks has version 20.03 for Windows and Linux distributions. The page also includes helpful notes for installation.

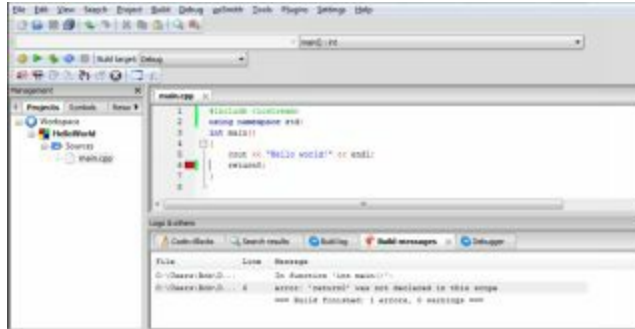
Once the program is installed, you are set. You can use it to write C++ programs. To do so you have to follow three steps: creating a file, building the program, and running the program.

Creating a new file is a bit more nuanced. Here is what you have to do:

1. Go to File > New
2. Select “Create an empty file” and input your code
3. Save the file with a “.cpp” extension

And, needless to say, you have to build a program before you can run it. You can do this by going to Build > “Build and Run” in the menu.





Screenshot of “Hello World” in Code::Blocks

This is a live screenshot of our "Hello World" program in code blocks. After this program is run, an error shows up in the Build Messages of the code. Code::Blocks highlights the error on line 6. Given your knowledge of C++ syntax from the earlier section, you should see how to repair the code. How can we repair the code?

Hint: Data and functions are supposed to show up in blue.

By the way, this program is very lightweight for the amount of work it allows you to do. Code::Block is available on Windows and Linux, but there is no Mac version.

Mac OS IDE Installation and Setup

For Mac, you will need to get **Xcode**. It is a free IDE software development suite for macOS. You can get it on the Mac App Store. Xcode supports a variety of other languages like Java, Python, and Ruby.

It is geared toward developing software for macOS operating systems; this can be for the tvOS for AppleTV, watchOS for Apple watches, and the iPadOS for iPad tablets (“What’s New in Xcode 9,” n.d.). Xcode offers a variety of **software development kits** (SDKs) for different MacOS platforms to help programmers through Apple’s proprietary programming schemes. SDKs are a set of tools, provided by hardware and software vendors, used for developing applications for specific platforms (“What’s New in Xcode 9,” n.d.). SDKs allow developers to be fully integrated within a development community, like Apple’s or Android’s.

Mac uses proprietary compilers that will only work with Xcode, so if you are on Mac you have little choice but to develop with C++ using Xcode. You can download and install Xcode by following this link:

<https://apps.apple.com/us/app/xcode/id497799835?mt=12>

Once the file is downloaded and installed, open Xcode and click on the “Create a new Xcode Project” icon. Then select “New Project” in the initial window.



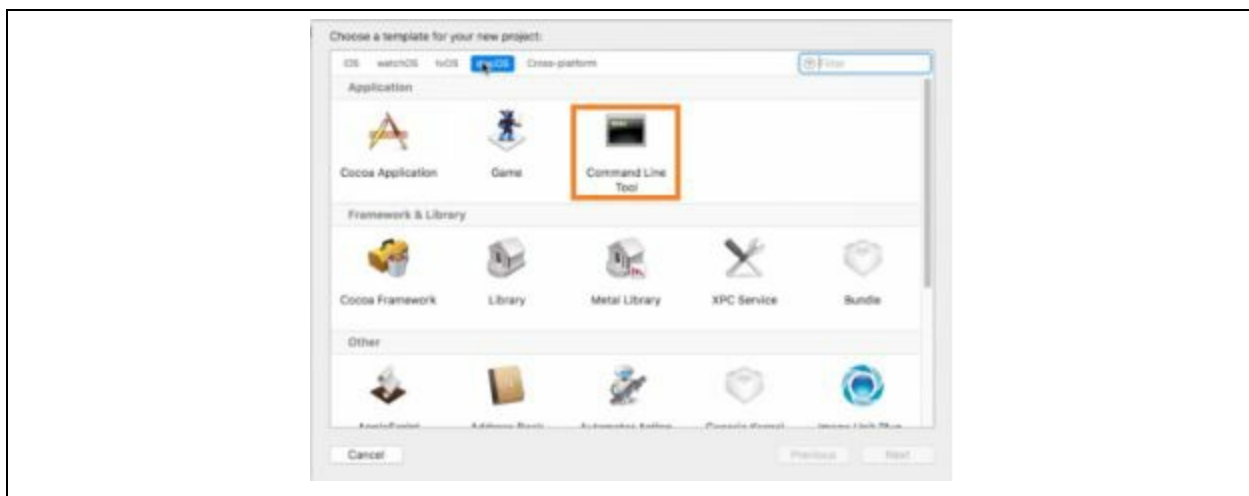
Initial Screen for Xcode: Select a new project

These screenshots feature an Xcode build from version 8.3.3 (Patel, 2017). Therefore, our examples might look different from a more up-to-date build. As of this writing, the most current version is 12.1, which mostly includes

updates to the various macOS platform SDKs (“Xcode,” n.d.). Despite these differences, these screenshots will still help you orient yourself in the more current build. Our instruction will still yield results.

After you do this, a prompt window will appear that will ask you to choose a template. This window will guide you through the rest of the setup. Select the macOS sections and go to the Application section; in there, choose the Command Line Tool.

It looks something like this:



1. Project Selection Screen: macOS > Application > Command Line Tool

Xcode has many built-in code utilities for running programs on various Apple platforms. Therefore, to run a test program you would have to select macOS for it to run on your program. Further, to access assembly language software development needed for C++, you would have to use Xcode’s command-line tool. This command-line tool can handle Objective-C and C languages as well.

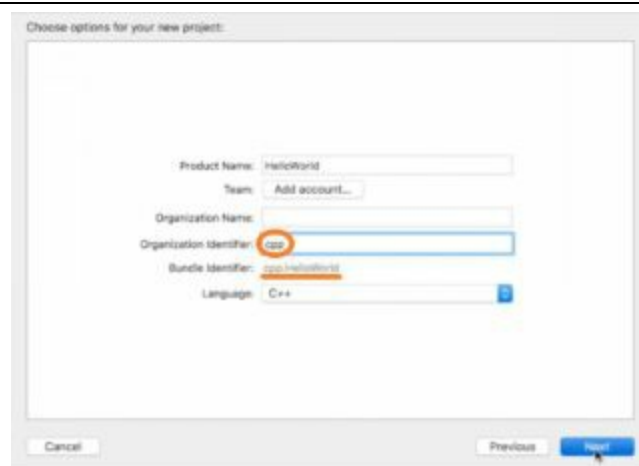
After you select a template, the guide will present options for the command-line tool. Here, you can name the project, add your organization’s name, use a bundle identifier, and select a programming language.



2. Command Line Tool Options: Language > Select C++

Xcode's command-line tool can handle Microsoft's proprietary languages like Swift. This program can run alongside C-based languages in Mac compilers, including C++. Swift is selected by default.

In the same window, you will be required to add an organization identifier. Remember that Xcode's command-line tool is a programming template for proprietary macOS operating systems, so the organization identifier is used to create a unique idea for your program in Apple databases: the Apple Developer Website and iCloud Container, iTunes connect portal in the Appstore. This will streamline your app into a proprietary macOS framework.



3. Command Line Tool Options: Organization Identifier > “cpp”

The organization's identifier is a naming convention that helps programmers integrate their projects into Apple's proprietary software development scheme. Here, we are using “cpp” as our identifier, which generates “cpp.HelloWorld” as our bundle identifier. Once we indicate an identifier, the guide allows us to proceed to the next step.

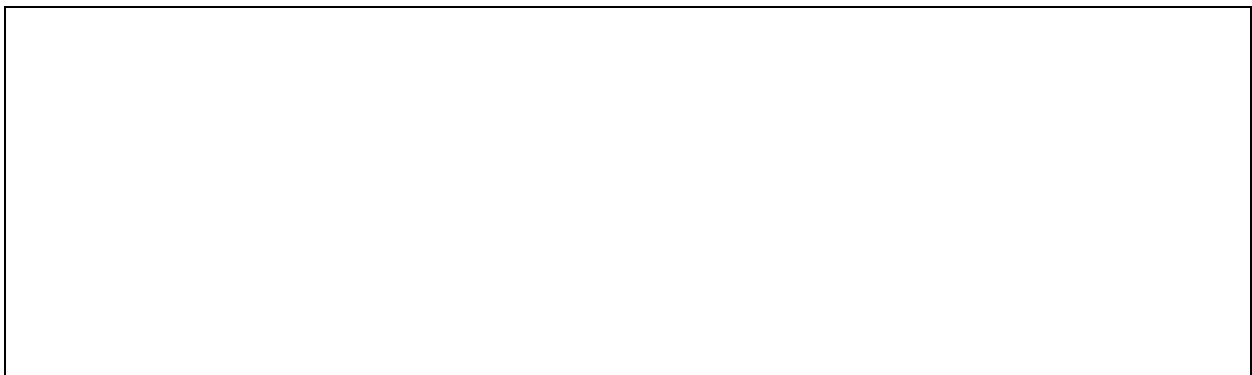
Now, save your project in a directory of your choice. On the left-hand side pane, Xcode will be able to open the file. You will have to select main.cpp to access the global C++ file for your program in Xcode. If you so desire, you can upload cpp files into the directory, and edit them.



3. Xcode Command Line Tool: “main.cpp”

This screenshot features the IDE style Xcode Command Line Tool workspace. It has the *main.cpp* opened in the text editor (Patel, 2017).

Once you have written some code and you want to run it, you select Product and click on Run. Also, you can run and build the program by selecting the button.



Linux Compiler Installation

Linux is a console-based operation system, so it does not need a text editor. Instead, we will focus on installing and setting up a programming compiler. Mac and Windows automatically initialize a compiler within the IDE, but in Linux, a more hands-on approach is required. You will be required to execute a backend initialization. It might seem like a lot of work, but in the great scheme of things, it allows those working with Linux to have full control of the development procedure and unlock more computing power.

Because Linux is an operating system that uses a console interface, it will be strange to beginners or anyone who is used to Windows or Mac. This is because users are often used to a GUI. The GUI-lessness of Linux allows more computing power to be freed for programming – a GUI takes some computing power to produce and maintain. This is why many servers and other programming environments have console interfaces. Also, many of them use **Linux distributions**.

A Linux distribution (distro) is an operating system made from Linux kernel-based software collection and package management system for installing additional software.

The following are instructions for installing and initializing a compiler. This will be for Ubuntu, a popular Linux distro for beginners. These installations will be similar in any Linux distro you will be using.

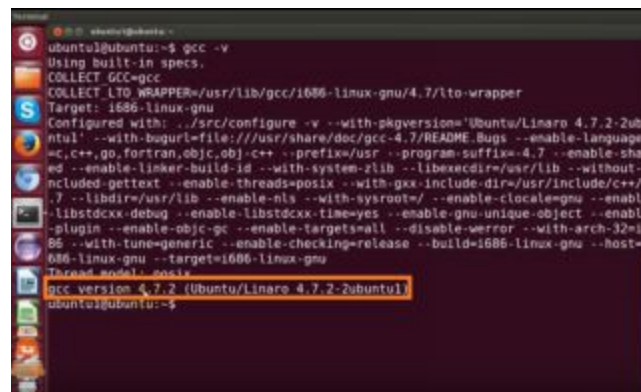
Note: There are many distributions of Linux that provide a GUI based desktop experience that many users love. Most of these are open-source and free to download and install. You can explore these distros and create console interfaces to practice on. One free distro to consider is **CentOS**, which is based on Red Hat for server management ([“Centos-faq | Open Source Community,”](#) n.d.). **Red Hat Enterprise Linux (RHEL)** is used in many server management setups and is not free. However, CentOS uses many of its components and is an excellent distro for preparing to work in Red Hat environments.

In this example, we will use the **GNU Collection Compiler** on Linux. The GNU Collection Compiler is a Linux-based compiler that supports C++ and other various languages like Fortran, Ada, and Java (“GCC 7 Release Series

—Changes, New Features, and Fixes—GNU Project—Free Software Foundation (FSF),” n.d.). To install GNU GCC, follow these steps:

1. In the console, enter the following commands:

Check first if there is a version already installed on your machine by entering this command: `gcc -v`



```
ubuntu@ubuntu:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.7/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.7.2-2ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-4.7/README.Bugs --enable-languages=c,c++,go,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.7 --enable-shared --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib --without-include-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.7 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin --enable-objc-gc --enable-targets=all --disable-werror --with-arch=32=i686 --with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1)
ubuntu@ubuntu:~$
```

A. Check for GNU GCC in Ubuntu Console

You should check your system for a copy of the GCC. This screenshot features a system with GCC already installed (Patel, 2014). After using the command, if there is no copy of GCC in your system, it will be missing this line. If there is a copy, it will have a similar line. Most likely, your version will be more recent.

The following commands will install GCC on your system (Agarwal, 2017b):

`sudo apt-get update`

`sudo apt-get install GCC`

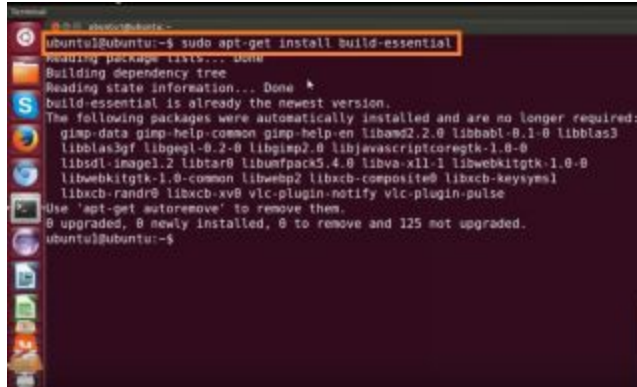
```
gcc version 4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1)
ubuntu@ubuntu:~$ sudo apt-get install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
gcc is already the newest version.
The following packages were automatically installed and are no longer required:
gimp-data gimp-help-common gimp-help-en libamd2.2.0 libbabl-0.1-0 libblas3
libblas3gf libbogl-0.2-0 libgimp2.0 libjavascriptcoregtk-1.0-0
libltdl-image1.2 libltdl0 libumfpack5.4.0 libva-x11-1 libwebkitgtk-1.0-0
libwebkitgtk-1.0-common libwebp2 libxcb-composite0 libxcb-keysyms1
libxcb-randr0 libxcb-xv0 vlc-plugin-notify vlc-plugin-pulse
Use 'apt-get autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 125 not upgraded.
```

A. Install GNU GCC in Ubuntu Console

In the screenshot, a GNU GCC is already present. Running the highlighted command will install the GNU GCC on your system if it isn't there (Patel, 2014). The console will prompt you to type "Y" for it to begin the installation. You also are offered package management features like the ability of the terminal to inform you of outdated or unnecessary packages. It will also suggest commands that will make your system more efficient. This explains the popularity of Linux with programmers; although wordy, it allows them to troubleshoot with ease, gives them more control, and it is easy to navigate when compared to GUIs. This is because programmers are more likely to remember a line of code than where something resides in the GUI.

The following command will install all the libraries required to compile code and eventually run C++:

`sudo apt-get install build-essential`



A. Install GNU GCC in Ubuntu Console

This screenshot shows a system with the build-essential libraries installed and inputting the highlighted installation command (Patel, 2014). Just as with installing the GCC, after using the command and finding there is no copy of the build-essential libraries in your system, it will prompt you to install. Type in “Y” and the terminal will install libraries.

2. Check the installation with the following command:

```
g++ --version
```

If all went well it will tell you what version of GCC is installed.

Because Linux has a built-in text editor you will have to use the following command to access the GUI for the text editor:

```
gedit
```

You will be free to write your program as you see fit. Remember to save the programs with the “.cpp” extension so that they will be compiled correctly.



Gedit in Ubuntu

This screenshot features Linux's built-in text-editor, Gedit ([Running C, C++ Programs in Linux] Ubuntu 16.04 (Ubuntu Tutorial for Beginners), n.d.). Gedit is a Linux programming text editor utility that has syntax highlighting and tabbed spacing features for a diverse array of programming languages. Be sure to have “C++” selected from the highlighted dropdown menu. You can save your program file in the GUI by clicking the “save” button. Be sure to list which directory you have your file saved, as you will have to point the compiler to that directory to compile and run the code.

To test and run your code you must follow these instructions:

1. Lead the terminal to the files directory

To do this use this command. Enter it in the directory repeatedly until the .cpp file is revealed.

`ls`

2. Compile and test the program file

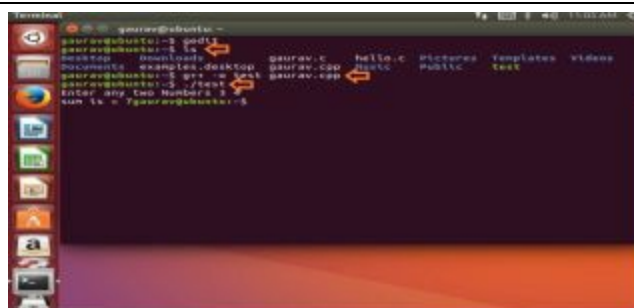
Use these commands to do so:

`g++ -o test[filename].cpp`

If there are any errors this command will tell you which line contains the error just like you would expect from an IDE. Then you can open the program file through “gedit [filename.cpp]” and fix the line of code.

3. Run the program file

The compiler will create an executable file called “test” that will run the program. You can execute it by entering “. ./test” in the terminal.



Testing, Compiling and Running a .cpp file in Ubuntu

This screenshot features the terminal testing, compiling, and running the example code written in gedit (*[Running C, C++ Programs in Linux] Ubuntu 16.04 (Ubuntu Tutorial for Beginners), n.d.*).

Linux distros are verbose console-based systems and most of them can handle programming in any language through the terminal. Learning to work in the Linux environment is beneficial as it exposes you to server-side programming.

Note: The process for installing GCC and compiling on CentOS is similar. To install, you instead use “*yumgroupinstallDevelopmentTools*”. This command automatically installs the needed libraries as well. The steps for composing, compiling, testing, and running the program are all the same.

Creating your own programming environment helps prepare you for real-world scenarios in a way that online coding spaces cannot. In this book, we will be using an online IDE. This is because everyone will have the same learning environment with standardized outputs and interfaces. It simplifies things.

Chapter 2:

Basics of C++, Principles of Programming

Programming languages are like any other human language: they have structure, syntax, and rules. It is not enough to know the words of another human language. Knowing how to use them and how they function in that language is an integral part of communicating effectively in that language. Programming languages function the same way, but the communication they care about trying to help is one between man and machine. While machines, at least today, can never be as clever and capable as we are, there is a lot they can understand at a rate and with an efficacy that is alien to us. All programming is a set of instructions. These instructions can be rules, they can be actions you want the machine to perform, or something else. But all this communication is possible because we follow the rules of a programming language.

Programming rules, or syntax, are a list of delineating symbols used to communicate aspects of a program like functions and variables. Syntax is important to the compiler because it tells the compiler what instructions it should give to the machine.

Before we start, it is worth keeping a few things in mind. Unlike Python or other high-level programming languages, C++ does not resemble the language of humans quite easily. High-level languages like JS, Python, and C# are made this way because it makes them easy to read and manage. C++, despite being this way, is easier for machines to understand precisely because it is not so abstracted. This is why when working with C++ you only need a compiler and text editor, whereas languages like Python need their environments installed.

Principles of Programming

C++ has many ways of completing the same task. Just like with human languages, there are many ways of saying the same thing, but some ways are best suited for certain occasions and some aren't. C++ is the same way; some methods are great because they reduce program overhead. Many developers will want that because it increases the performance of the entire application, and that is the first principle of programming: complete a task with the least amount of functions as possible. Do not make things more complicated than they need to be. In most cases, this means using a loop or a switch statement in a frugal way.

The best way to do this is by understanding the nature of the problem first and how best to implement the solution. In other words, you will need to write an algorithm – a list of instructions that you want the machine to follow to fix the problem.

Your designs should always begin with the thing you want the program to do. This will also include your algorithm, or at least the problem that your algorithm will fix. The algorithm will always look something like this:

1. Input: data coming, where applicable
2. Processing: operations performed on the data and declaration of variables
3. Output: the results, or the action you want performed.

Here is a verbose example of a C++ program for displaying “Hello World”. This is to illustrate how C++ programs are composed:

```
// Simple C++ program to display "Hello World"
```

```
// Header file for input output functions
```

```
#include<iostream>
```

```
using namespace std;
```

```
// main function -
```

```
// where the execution of program begins
```

```
int main()
```

```
{  
    // prints hello world  
    cout<<"Hello World";  
    return 0;  
}
```

Here is the structure:

1. Call header file for input and output functions
2. Call the main function where the execution happens
3. Print “Hello World”
4. Terminating statement that indicates the state of the machine

We can see how verbose C++ is, because most of the steps regard the backend aspects of C++ that are needed to execute the code, such as calling the library files. Your algorithms don’t need to be this detailed most of the time.

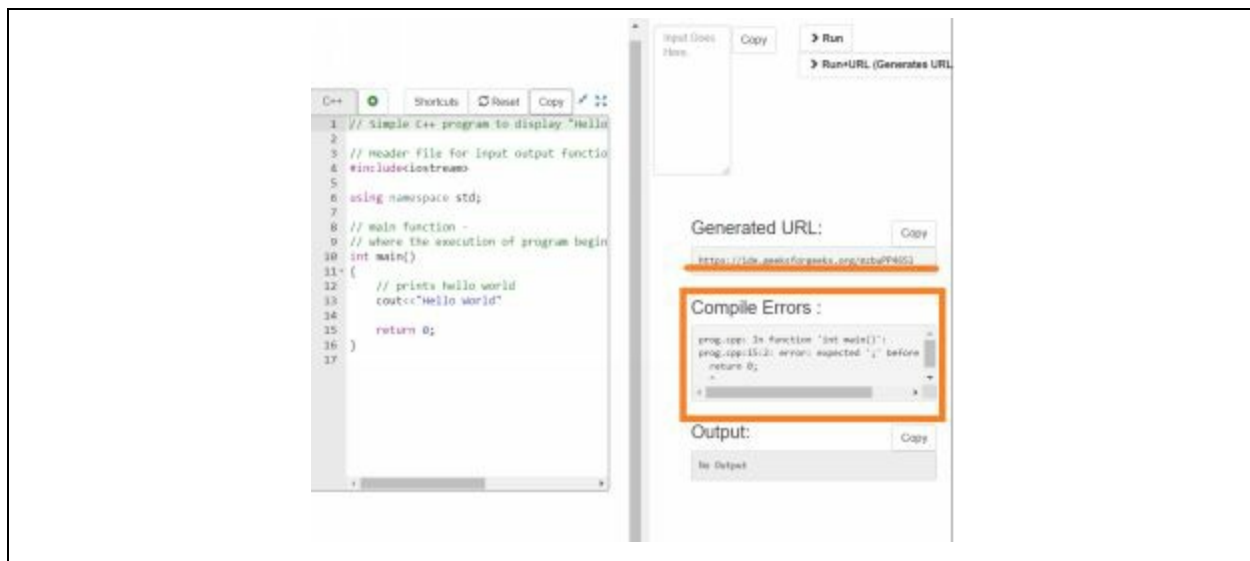
Despite being simple, this algorithm illustrates how developers take tasks and translate them into code. In the next chapter, we will use this algorithm to write our program.

As you have noticed in the example, there are comments in the code that explain what each part of the code does. These comments are preceded by “//”. The compiler will not read any lines of code preceded by // because they don’t do anything but help other people working on the code read what the code does. It is good practice to include comments in your code so people know what each line of code does.

Overview of the C++ Syntax

We have talked about the importance of syntax for the compiler, but it is also important to understand the syntax of a programming language because it will allow you to debug the code. Just like with human languages, if you know the rules of a language you can correct yourself easily when you make mistakes. **Debugging** is the process of finding and removing errors and abnormalities in the code, also known as “bugs” (“What is Debugging?” n.d.). It is a process of correction.

IDEs have debugging tools that highlight where errors occur in the code, but these features aren’t always reliable. For instance, leaving out a semicolon to terminate a line of code is a common error, but the debugger will highlight the error in the (.h) file instead of the code where the mistake happens in the program file (.cpp). Most debuggers are not intelligent enough to detect an absence of something and will throw the verbose error that the compiler gives when the unterminated line violates a rule in the library. However, a programmer will pick up the error simply when they notice the red semicolon line delimiter is missing from one of the lines.



“Hello World” program in GeeksforGeeks IDE without a line delimiter

This screenshot features “Hello World” with a missing delimiter. Immediately you will notice that the IDE has different syntax highlighting for C++ than the programming environments we explored in the previous chapter. Syntax can vary from platform to platform. This is why we are

standardizing the process with GeeksforGeeks IDE. In our chosen IDE, semicolons are not highlighted. However, the IDE can make up for this by recognizing and annotating missing semicolons, as highlighted in the screenshot. You can practice and explore this program by going to its generated URL: <https://ide.geeksforgeeks.org/mzbuPP46Sl>.

It is easy to see this when the code is small, but when working on a larger project the task can be difficult. I have heard of programmers who spend a week and or more trying to find or fix a line in their code only to discover it is something very minuscule like a missing letter or mismatch in case. To make debugging easier, it is advised that you separate code into smaller modules (“What is Debugging?” n.d.). Many errors can be found and easily dealt with in this way.

Let us look at the “Hello World” program and study its syntax in detail. While syntax highlighting may vary across different platforms, these aspects of code are consistently highlighted on all platforms.

Comments: As I said, these are the lines of code that the compiler will ignore. They are used to annotate code and leave helpful explanations for other programmers. This is great because normally when we code we collaborate with others. It advised that you leave comments in your algorithm to simplify it for others. In most platforms comments will be highlighted green. Comments are always preceded with (*//*).

Header File Library: Lines that start with (*#*) are used by the compiler to call library functions. These lines are very essential, so they appear in every program. In the “Hello World” example we call a library that is used to manage strings. These lines of code will be highlighted in different colors across different platforms. In GeeksforGeeks the color is purple.

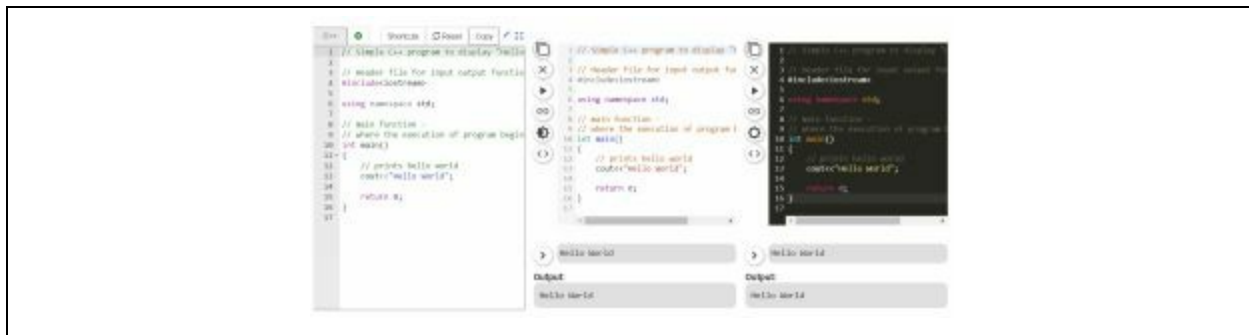
Statements: Statements describe the beginning of a line of code with instructions that the compiler recognizes. This includes declaration handling instructions, like “*using...*” or “*return...*” in our Hello World program. They are also highlighted differently

across multiple platforms. On GeeksForGeeks they are highlighted purple.

Functions: This is code that encapsulates instructions. It may take inputs and output a result. Every function is written like this: “[*function_name*] ()” followed by curly braces {}. Anything in the curly braces is the set of instructions that will be executed. In our example, *main()* is the function that initiates the instructions that will output “Hello World” with `cout<<`. In a “.cpp” program, *int main ()* must always appear for the program to **function**.

Note: Recalling our discussion about algorithms from the previous section, your algorithm will mainly describe what instructions occur within the curly brackets of *intmain()*. More detailed algorithms may explain the sorts of data types needed for inputs, outputs, and other data. We’ll discuss data types in detail in Chapter 9.

Data: These are variables and other kinds of data functions and statements. In the GeeksforGeeks IDE, they are highlighted blue.



Comparison of “Hello World” program in GeeksforGeeks IDE and the GeeksforGeeks Embedded Program Articles

This screenshot features a successful run of “Hello World” with the memory used to run the program. This IDE has light syntax highlighting, focusing mainly on comments, statements, and operand data. Compared to GeeksforGeeks’ embedded code in their articles, their embedded code is much more detailed ([“Writing first C++ program,” 2017](#)). The two screenshots on the right feature the “light-mode” and “dark-mode” views of the same code embedded in their articles. These views are more detailed, highlighting comments, functions, and distinguishing different statements.

For example, data types such as *int* are distinguished with a different color from other statements such as *using namespace*.

Also the embedded views, unlike the IDE, distinguish strings from other data. This is a perfect example of how syntax highlighting can vary across different platforms.

Earlier we saw a Code::Block example with an error in it. Below is the very same code next to one that is corrected. Immediately you can begin to appreciate the syntax and what went wrong in the Code::Block example.



Comparison of “Hello World” in Code::Blocks with GeeksforGeeks IDE

This screenshot features a side-by-side comparison of the previous example, “Hello World” in Code::Blocks with errors, and the same corrected code in GeeksforGeeks IDE. Upon studying both, you should be able to see that the last *int main()* instruction in the function was incorrectly entered in the Code::Blocks. As a result, the compiler could not recognize the *return* statement.

I will admit that it is not as impressive in a few code lines like this, but being able to notice errors and explain them is one of the most important skills for programmers. At this point you should not worry much about being able to do this. The more you code and learn, the better equipped you will be to notice and fix errors. Software and web development are industries with a high failure rate because of this. To succeed as a developer you need to solve problems quickly, find bugs more efficiently, and fix them. This also applies on the administrative side, because being able to find, fix, and explain bugs is

a large part of desk service level agreements.

In tech, **Service Level Agreements** (SLAs) are agreements between a service provider and a client. They agree upon aspects of the service, like support, quality, availability, and responsibilities (Wieder, Butler, Theilmann, & Yahyapour, 2011). Comprehensive SLAs offer a debugging debriefing, where every error is logged and listed with its cause, workaround, and solution. This is the best way to do it in technology management firms and development stages. However, administrators have more resources to dedicate to debugging than developers themselves. They are compensated at a higher rate as well. For instance, web developers make approximately \$60,000 a year in the US; web administrators, those in charge of SLAs, make about 1/3 more at \$90,000 a year (“15-1199.03—Web Administrators,” n.d.). This level of expertise is valuable because it understands how the compiler translates C++.

First Program: Output and Basic Strings

“Hello World” is every programming language’s beginner project; its aim is to show the learner the basic syntax of the program and how it functions. As you learn more languages you will write many “Hello World” programs. In the previous section, we used it to explore topics like algorithms, syntax, and C++ programming environments. In this section we will go further: we will manipulate the code.

C++ uses the *cout* object from the *iostream* library. An object is a method that a computer uses to manage data. In our example, *cout* was used to tell the compiler to print “Hello World” with the `>>` operator. Don’t freak out; we will discuss operators and objects in later chapters. For now, it’s enough to understand that the compiler reads *cout* as specific instructions because that instruction is defined in the *iostream* library that we declared.

Now, let’s look at “Hello World” and the *cout* object by changing the print out. Find the original code here so you can use it as a reference: <https://ide.geeksforgeeks.org/eA8ZMEKiDO>



“Hello World” program in GeeksforGeeks IDE

This screenshot is the successful run of “Hello World” in split-screen mode. As we begin to use the IDE for coding exercises, note the input and output boxes. While there are no inputs for this program, you will input data into the top box labeled “Input Goes Here...” when prompted by the program in the output below.

You can access this program in the saved IDE here: <https://ide.geeksforgeeks.org/eA8ZMEKiDO>.

You can also view a copy of the code in the index.

Using Cout

The *cout* object uses the << operator to print values and text, and we can have as many of them as we like. Using the IDE in your browser, change the code to print out:

Hello World

I am learning C++

Like this:



The screenshot shows a C++ IDE with a code editor on the left and a console on the right. The code in the editor is as follows:

```
1 // Simple C++ program to display "Hello World"
2
3 // Header file for input output functions
4 #include <iostream>
5
6 using namespace std;
7
8 // main function -
9 // where the execution of program begins
10 int main()
11 {
12     // prints hello world
13     cout<<"Hello World!";
14     cout<<"I am learning C++";
15
16     return 0;
17 }
18
```

The lines 13 and 14 are highlighted with an orange box. The console on the right shows the output:

```
Generated URL: https://ide.geeksforgoeks.org/ed62961d0
Time(sec): 0 Memory(MB): 3.3542941842651
Output: Hello World! I am learning C++
```

Adding cout objects

This screenshot is the successful run of our edited “Hello World” program in split-screen mode. While the program ran without any errors, it didn’t produce the result we wanted. Observing the output, the printed text is missing a new line. We will have to instruct the compiler to add the new line to get the desired result.

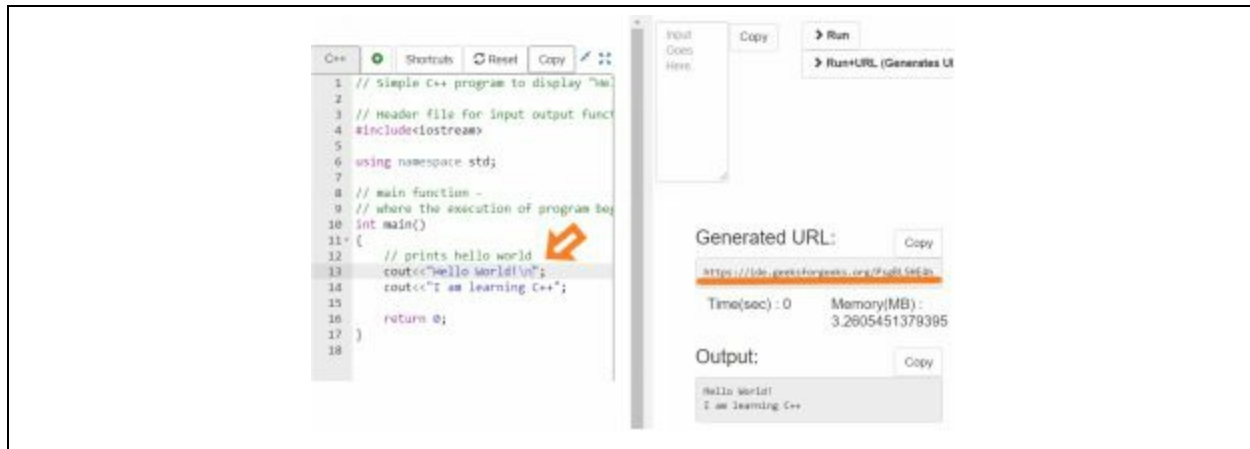
Adding lines can be done in several ways: we can add delimiters within the strings or use another manipulator object from the *iostream* library.

Using Escape Sequences

Although it might be tempting to write a string within *cout* and enter a line, it will not work. It will break the code and it will not run. Don’t believe me? Test it!

You can add a new line by using an escape sequence. **Escape sequences** are

used in many instances where you need to introduce special characters within strings and character streams. It can also add lines, like so:

A screenshot of a C++ IDE interface. The main editor window displays a C++ program with the following code:

```
1 // Simple C++ program to display "Hello World"
2
3 // Header file for input output functions
4 #include <iostream>
5
6 using namespace std;
7
8 // main function -
9 // where the execution of program begins
10 int main()
11 {
12     // prints hello world
13     cout << "Hello World!\n";
14     cout << "I am learning C++";
15
16     return 0;
17 }
```

An orange arrow points to the `\n` escape sequence in line 13. To the right of the editor is a sidebar with buttons for 'Input Goes Here', 'Copy', 'Run', and 'Run+URL (Generates URL)'. Below these buttons, the 'Generated URL' is shown as <https://ide.geppcforgeeks.org/?fullScreen>. Below the URL, the execution statistics are displayed: 'Time(sec): 0' and 'Memory(MB): 3.2805451378395'. At the bottom, the 'Output' section shows the program's output: 'Hello World!' followed by a new line and 'I am learning C++'.

Using in-string escape sequences to delimit and manipulate the output

This screenshot is the successful run of our edited “Hello World” program in split-screen mode. Using `\n`, we were able to get our expected output.

As you look at the code, you can see that doubles quotes are used. So, if you want to add double quotes in a string, you will have to use an escape sequence so that the compiler knows that is not a part of the code. Not doing so may cause unforeseen errors or bugs. This is why escape sequences matter. Here is a list of the many instances where they can be used (“Escape sequences—Cplusplus.com,” n.d.):

“`\'`”: Used for single quotes

“`\"`”: Used for double quotes

“`\?`”: Used for question marks

“`\\`”: Used for backslashes

“`\f`”: Stands for “form feed” and is used to go to the next “page”

“`\n`”: Stands for “line feed” and is used to go to the next line

“`\t`”: Stands for “horizontal tab” and adds 5 spaces horizontally

“`\v`”: Stands for “vertical tab” and is used for spacing in vertical languages

Note: If you decide to try different escape sequences as text delimiters, be sure to only use the *Run* button. This will save the workspace. If you want to save some of your results for later, you can use the *Run + URL...* button to save your work at another URL to return to later. To return to our exercise, you can use our saved IDE workspace URL (<https://ide.geeksforgeeks.org/PsgBLSHE4h>). You can also access this code in the index.

Using Endl: Input/Output Manipulators

Like we have said, programming languages are as versatile as human language. You can express one instruction in many ways, and terminating a string is no exception. The difference between an escape sequence and *endl* is that *endl* flushes and refreshes the buffer used to store the string. Programmers use *endl* to clear memory after printing a lot of text. You can see why this is useful in memory management. *Endl* is an example of an input/output manipulator. **Input/Output manipulators** are functions that allow you to control input/output streams using the “<<” operator or the “>>” operator (“Input/output manipulators—Cplusplus.com,” n.d.).



The screenshot displays a C++ IDE interface. On the left, a code editor shows a program that prints 'Hello World!' followed by 'I am learning C++' on the same line. The code uses `cout<<"Hello World!"<< endl;` and `cout<<"I am learning C++";`. An orange arrow points to the `endl` in the first line. On the right, the IDE's output panel shows the generated URL, execution time (0 seconds), memory usage (3.23 MB), and the program's output, which is 'Hello World!' followed by 'I am learning C++' on the same line.

```
1 // Simple C++ program to display "Hello World!"
2
3 // Header file for input output function
4 #include<iostream>
5
6 using namespace std;
7
8 // main function -
9 // where the execution of program begins
10 int main()
11 {
12     // prints Hello World!
13     cout<<"Hello World!"<< endl;
14     cout<<"I am learning C++";
15
16     return 0;
17 }
18
```

Generated URL: <https://ide.geeksforgeeks.org/tocFFytc71>

Time(sec): 0 Memory(MB): 3.2334406854956

Output: Hello World!
I am learning C++

Using an input/output manipulator to delimit and manipulate the output.

This screenshot is the successful run of our edited “Hello World” program in split-screen mode. Using *endl* we were able to get our expected output. In

a program this small, it is difficult to see the effects of using a memory manipulator. Manipulators only begin to make a difference when using them in called functions. You can access the code at this URL: <https://ide.geeksforgeeks.org/htcfFytK71>.

Like escape sequences, there are many more input/output manipulators that give programmers more control over data types or strings. Many of them will make sense once we talk about data types and structuring classes. Here are some of them (“Input/output manipulators—Cplusplus.com,” n.d.):

`“boolalpha/noboolalpha”`: Switches between using “0/1” to “false/true” for Boolean values

`“showbase/noshowbase”`: For mathematical outputs, controls whether a prefix is used to indicate a numeric base

`“showpos/noshowpos”`: Controls whether the “+” sign is used to indicate non-negative numbers

`“uppercase/nouppercase”`: Controls the usage of uppercase characters with particular output formats

`“ends”`: Outputs “\0”

`“flush”`: Flushes the output stream

`“endl”`: Outputs “\n” and flushes the output stream

`“get_money”`: Receives an input as a monetary value

`“put_money”`: Formats and outputs a monetary value

`“get_time”`: Receives an input as a date/time value according to a specified format

`“put_time”`: Receives an input as a date/time value according to a specified format

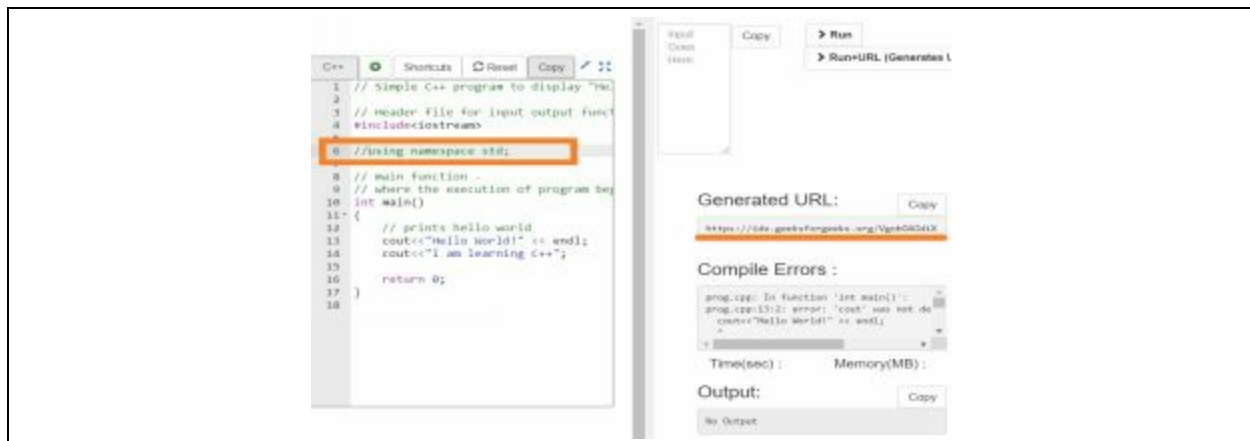
Note: Recall our conversation about C++11 and C++14. As of this writing, most programming IDEs use C++11 as their default version. The “get/put...” functions are all in C++11. If you receive an error trying to use these manipulators, check to see if your setup is using C++11. Additionally, `“quoted”` is a manipulator that is in C++14 only. This

manipulator allows you to insert and extract quoted strings with embedded spaces ([“Input/output manipulators—Cplusplus.com,” n.d.](https://www.cppreference.com/w/cpp/string/basic/basic_string_manipulator)).

Omitting Namespace

We have already spoken about using *namespace* statements. We said it helps declare strings globally, but when it comes to a small program like “Hello World” it is not so important. For larger programs that use several functions, it is good practice to use the statement to declare each string separately. This saves memory space in the compiler, or you will find yourself in a situation where the compiler is bogged down unnecessarily, having to pull the entire string namespace library to print code. Declaring each string separately might make your code wordier, but it will reduce compiler overhead, and it focuses your instructions. *Std namespace* has multiple objects and definitions that can make it difficult for the computer to find the appropriate way to manipulate and define a string. The namespace convention is important when you have functions that are going to be called by multiple programs. So putting it in there unnecessarily has the potential of making your functions poorly defined and causing recognition errors between them.

Let’s make our code more efficient by removing the using *namespace* from our code.



Testing Stability: Using a comment to remove “using namespace”

This screenshot is an unsuccessful test run of the “Hello World” program with `using namespace std;` removed. You can access this saved IDE space through the following URL: <https://ide.geeksforgeeks.org/VgobOAOdiX>.

“Commenting out” lines of code is a highly efficient way of testing the

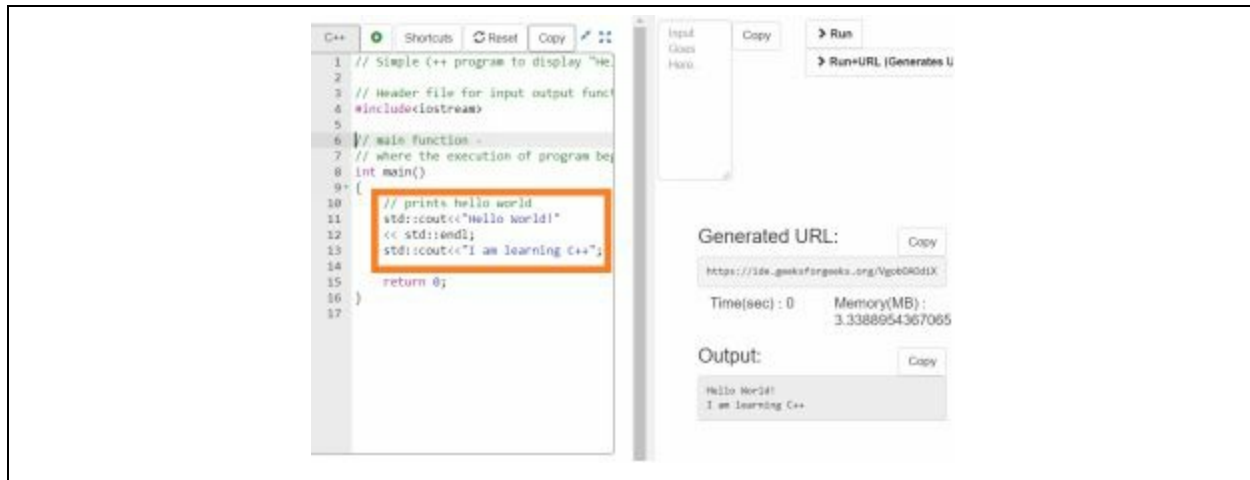
stability of code. You can delete the double forward slashes to remove the comment and restore the line of code at any time. This allows you to test the code without destroying it. Once you have completed testing, it is good practice to delete any unwanted lines of code. During your testing, you can always return to this saved instance by using the URL above. You can also access the raw code in the index.

When you remove *using namespace std* the following error occurs:

```
prog.cpp: In function 'int main()':
prog.cpp:13:2: error: 'cout' was not declared in this scope
  cout<<"Hello World!" << endl;
  ^
prog.cpp:13:2: note: suggested alternative:
In file included from prog.cpp:4:0:
/usr/include/c++/5/iostream:61:18: note: 'std::cout'
  extern ostream cout; /// Linked to standard output
                  ^
prog.cpp:13:26: error: 'endl' was not declared in this scope
  cout<<"Hello World!" << endl;
                        ^
prog.cpp:13:26: note: suggested alternative:
In file included from /usr/include/c++/5/iostream:39:0,
                  from prog.cpp:4:
/usr/include/c++/5/ostream:590:5: note: 'std::endl'
  endl(basic_ostream<_CharT, _Traits>& __os)
  ^
```

Looking closely, the compiler tells us that we need to declare *cout* and *endl*.

It even makes suggestions, like declaring them as *std*. Let's do as the compiler asks and see what happens.

A screenshot of an online IDE. On the left, a C++ code editor shows a program with line numbers 1 to 17. The code includes a comment, a header file, a main function, and two cout statements. The second cout statement is highlighted with an orange box. On the right, there's a 'Run' button and a 'Generated URL' section. Below that, the execution time and memory usage are shown. At the bottom, the output of the program is displayed.

```
1 // Simple C++ program to display "Hello World!"
2
3 // Header file for input output functions
4 #include <iostream>
5
6 // main function
7 // where the execution of program begins
8 int main()
9 {
10     // prints hello world
11     std::cout<<"Hello World!"
12     << std::endl;
13     std::cout<<"I am learning C++";
14
15     return 0;
16 }
17
```

Generated URL: <https://ide.geeksforgeeks.org/VgobOAODiX>

Time(sec): 0 Memory(MB): 3.3388954367065

Output: Hello World!
I am learning C++

Declaring the iostream objects and manipulators

This screenshot is a successful test run of the “Hello World” program with *using namespace std*; removed. Once the iostream objects and manipulators were successfully declared, the unneeded line was removed.

If you would like to practice, you can use this URL: <https://ide.geeksforgeeks.org/VgobOAODiX>

Note: If you are having multiple errors during your practice, we will always provide a saved IDE workspace with a successful run of the program to help you troubleshoot. To begin the troubleshooting session, open up another window, and navigate to the successful run. Compare your current workspace with the successful run workspace. You can find a successful run of “Declaring the iostream objects and manipulators” at this link: <https://ide.geeksforgeeks.org/a5zOdSWoQU>.

Chapter 3:

Variables and Data Types

We have talked about how low-level programming closely resembles machine language, but they differ because they still use elements that are reminiscent of human languages like syntax, structures, and variables. **Variables** are a mathematical concept that humans use to convey meaning to machines. A huge chunk of computer science is recognizing that speech is a series of mathematical concepts. This includes AND and OR gates that enable machines to make decisions on a single data bit level. At a very small scale, bitwise functions can help with many computing processes, from controlling displays to switching traffic on a server. We will discuss these advanced topics later in Chapter 4. For now, let's deepen our understanding of C++ by looking at variables.

Variables are one of the most important aspects of any programming language - functions, and other operations of a programming language, generally operate on variables. We have said that not all programs need input or output, but all programs will have some variable that they are working on or there is nothing to work on. The outcome of a function is always a state change. A state change is when a machine changes from one decision or operational phase to another. This term was used to describe operations in old mainframe computers, but we still use it in computer science to describe how programs work. In our example, the IDE started with an initialized *main()* state before the program ran. Then it switched states to execute the output command.

State changes are discussed often in lower-level languages because of how close they are to machine language. Lower-level languages are also called **compiled languages** because they are compiled into machine language. This is in contrast to high-level languages like Python. If you remember, we spoke of how higher-level languages do not need variables to be declared with their data types because of their built-in libraries. For instance, Python uses a

Python environment to recognize variables and their data types. In C++ variables and the data types have to be declared. A **data type** tells the compiler how to treat the data. Data types make more sense when you think of vectors in math; unlike integers, which communicate amounts, vectors tell us the size and direction of space. Therefore, vectors are a different data type than an integer. Properties like these are used in software like Photoshop to help with rendering.

Note: Vector-based images can be manipulated without losing the quality of the image. This is because files like .jpeg or .png store pixel location of an image as integers on a grid. Vector files such as svg or .pdf use vectors to describe their pixel locations. Therefore, when the image is made larger, the vectors indicate how the colors should expand while the regular image files do not. This is what causes the pixelation and distortion when manipulating non-vector image files.

Input and Output: Declaring Variables

In the previous chapter, we declared some things as we were learning how to use *cout*. In this section, we are going to look at examples of data types and how they are declared. Below are examples of variable declarations and what they stand for (“C++ Variables,” n.d.). Keep in mind that variables essentially store types of data:

“*int*”: Keyword is short for “integer” and stores whole numbers, without decimals. This keyword includes positive and negative integers.

“*double*”: Stores numbers with decimals, both positive and negative numbers.

“*char*”: Keyword is short for “character” and stores single characters regardless of their capitalization. Char values are single quoted.

“*string*”: Stores text, such as in our "Hello World" example. String values are double quoted.

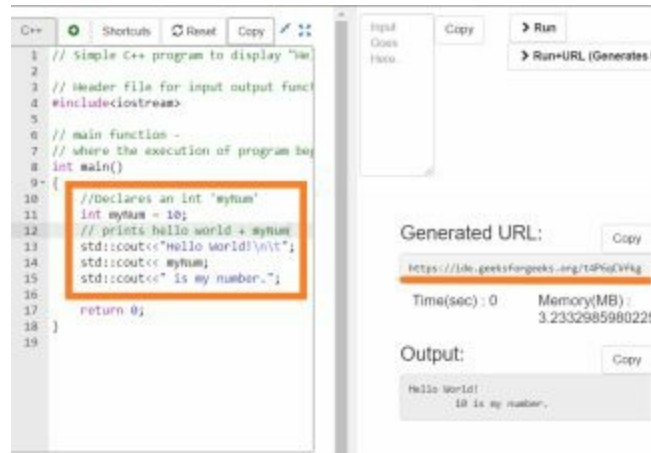
“*bool*”: Keyword is short for “Boolean” and stores values with two states: true or false. As previously discussed, these values can be expressed as either “0/1” or “false/true.”

In the next section, we will discuss them in detail. Now let us look at examples of these. To create a variable you have to assign it a value after the “=” operator. Their syntax looks like this:

[Datatype][Variable_name] = [value];

Note: Each *[Variable_name]* must be unique within your function. Otherwise, it will cause an error.

Now let us create a variable called “*myNum*” with the *int* data type and a value of 10, then have our program print it out.

A screenshot of an IDE window. The left pane shows C++ code for a program named 'myNum'. The code includes a header file, a main function, and a variable declaration. The right pane shows the execution results, including a generated URL, execution time, memory usage, and the program's output.

```
1 // Simple C++ program to display "Hello World"
2
3 // Header File for input output function
4 #include <iostream>
5
6 // main function -
7 // where the execution of program begins
8 int main()
9 {
10     //Declares an int 'myNum'
11     int myNum = 10;
12     // prints hello world + myNum
13     std::cout << "Hello World!\n";
14     std::cout << myNum;
15     std::cout << " is my number.";
16
17     return 0;
18 }
19
```

Generated URL: <https://ide.geeksforgeeks.org/t4P6qCVfkg>

Time(sec) : 0 Memory(MB) : 3.2332985980226

Output:

```
Hello World!
10
10 is my number.
```

Declaring int numbers

This screenshot is a successful test run of the “myNum” program. You can access the saved IDE workspace with the following URL: <https://ide.geeksforgeeks.org/t4P6qCVfkg>.

This code was implemented without `using namespace std;` as it's the best practice. You should practice implementing the same code successfully with the namespace. Remember that you can reset the workspace to a successful state by using the URL.

We had our share of `cout` examples in our code. Let's use the input box in the top right and the `cin` object so our program takes in inputs and operates on them.

We can do this by writing programs that take two numbers and adds them together. To do this, we have to declare two variables that will store our inputs for our program to operate on.

We will declare multiple variables:

1. Two *int* variables named *x* and *y*,
2. One *int* variable named *sum*.

We can also tell the program to prompt the user to give two inputs.

Note: In our IDE, you must enter the inputs *before* you run the program. Each input should be on its own line.



The screenshot displays a C++ IDE with a code editor on the left and a console/output area on the right. The code in the editor is a simple program that prompts the user for two numbers, calculates their sum, and prints the result. The code is as follows:

```
1 #include<iostream>
2 int main()
3 {
4     // Explains program to user
5     std::cout<<"Welcome! Enter two numbers\n";
6     //Declares input variables
7     int x, y;
8     //Declares output variable
9     int sum;
10
11     //Prompts user for inputs
12     std::cout<<"Type a number: ";
13     std::cin >> x;
14     std::cout << x <<"\n";
15     std::cout << "Type another number: ";
16     std::cin >> y;
17     std::cout << y <<"\n";
18     //Operates on the variables
19     sum= x+y;
20
21     //Prints results
22     std::cout<<"Sum is: "<<sum;
23     return 0;
24 }
25
```

The right side of the IDE shows the execution results. It includes a "Generated URL:" field with the value <https://ide.geeksforgeeks.org/xDXaHjutDO>, a "Time(sec): 0" and "Memory(MB): 3.2496786764526" section, and an "Output:" section containing the following text:

```
Welcome! Enter two numbers and
I will sum them.
Type a number: 11
Type another number: 12
Sum is: 23
```

Summing inputs and printing the result

This screenshot is a successful test run of our “Summing” program. You can access the saved IDE workspace with the following URL: <https://ide.geeksforgeeks.org/xDXaHjutDO>. This exercise had to be modified for our IDE’s input system. Therefore, we included additional `cout` lines that confirm our inputs.

This code was implemented without `using namespace std;` as it's the best practice. You should practice implementing the same code successfully with the namespace. Remember that you can reset the workspace to the successful state by using the URL.

Basic Data Types

Data types specify the size and types of information stored in a variable. Each has its own keyword and an associated size (“C++ Data Types,” n.d) :

“*int*”: Keyword is short for “integer” and stores whole numbers. Has a size of 4 bytes.

“*float*”: Stores decimal numbers with 7 decimal digits. Has a size of 4 bytes.

“*double*”: Stores decimal numbers with 15 decimal digits. Has a size of 8 bytes.

“*char*”: Keyword is short for “character” and stores single characters regardless of their capitalization. Has a size of 1 byte.

“*bool*”: Keyword is short for “boolean” and stores values that have a state of either being true or false. Has a size of 1 byte.

Note: Recall the “*string*” data type. We have worked with this special datatype extensively. String values can vary in size. Due to this, they are handled in the `iostream` library to apply more flexibility for text handling. We’ll return to strings and discuss some advanced topics in the next section.

These sizes matter for data handling. For instance, when you use *int* with *float/double* all those numbers will be converted to *int*. It is because the compiler uses 4 bytes to calculate integers which have the allocation for decimals. The program will still run but your calculations will not be accurate. Let’s explore this more by modifying our program and looking at the results. We are going to be using a long-popular decimal number (pi):

$\frac{22}{7} = 3.1428571428571428571428571428571$. It will test the limits of *float* and *double*.

```
1 #include<iostream>
2 int main()
3 {
4     // Explains program to user
5     std::cout<<"welcome! enter two num"
6     //Declares input variables
7     int x,y;
8     //Declares output variable
9     float result;
10
11     //Prompts user for inputs
12     std::cout<<"Type a number: ";
13     std::cin >> x;
14     std::cout << x <<"\n";
15     std::cout << "type another number: ";
16     std::cin >> y;
17     std::cout << y <<"\n";
18     //Operates on the variables
19     result= x/y;
20
21     //Prints results
22     std::cout<<"Result is: "<<result;
23     return 0;
24 }
```

Generated URL: <https://ide.geeksforgeeks.org/a8Mw6tmyf8>

Time(sec): 0 Memory(MB): 3.3904097859302

Output:

```
Welcome! Enter two numbers and
I will divide them.
Type a number: 22
Type another number: 7
Result is: 3
```

Using Int with Float or Double

This screenshot is a successful test run of our modified program calculating

π

$= \frac{22}{7}$. You can access the saved IDE workspace with the following URL:

<https://ide.geeksforgeeks.org/a8Mw6tmyf8>

. This exercise also includes $\frac{22}{7}$

$= 3.1428571428571428571428571428571$

...

but as you can see, the result is shown as 3. This is because the compiler calculated the inputs as *int*

, which are not modified to store decimals. As a result, the result is missing the decimals.

To get the result we are looking for, it would be best to change the variable data types. Will using float data types differ from double data types? Can we expect a different result? Remember you can reset by going to a successful state of the workspace using the URL.

```
1 #include<iostream>
2 int main()
3 {
4     //Declares operation variables
5     float x,y;
6     double xd,yd;
7     //Declares output variable
8     float result;
9     double resultd;
10     bool test;
11
12     //Operates on the variables
13     result= x/y;
14     resultd= xd/yd;
15     test=(result==resultd);
16
17     //Prints results
18     std::cout<< "True or False? Is float" <<
19     std::cout<< " is equal to double?" << xd <<
20     std::cout << std::boolalpha
21     << "!"<<test;
22     return 0;
23 }
```

Generated URL: <https://ide.geeksforgeeks.org/8K18P805W>

Time(sec): 0 Memory(MB): 3.4138678857422

Output:

```
True or False?
float(22/7) is equal to double(22/7)
False
```

Comparing Float and Double

This screenshot is a successful test run of our modified program calculating $\pi = \frac{22}{7}$. To focus on comparing *float* and *double*, we have removed the input statements and rearranged our data types. We highly suggest creating this program for yourself using the previous program located at this URL: <https://ide.geeksforgeeks.org/a8Mw6tmyf8>. This exercise will give you practice in some previous coding topics we discussed, including:

1. “*bool*” data type keyword
2. “*boolalpha/noboolalpha*” input-output manipulator

It will also give you some experience with a logical operator. We will discuss the logical operators in detail in the next chapter. Once you have manipulated the program, open a new window, and compare this to the successful run of the screenshot above. You can find the saved IDE workspace with the successful run at this URL: <https://ide.geeksforgeeks.org/dNIUdYtN7M>.

Power of C++: Advanced Strings

Another data type we are going to look at is strings. **Strings** are one of the most popular data types in computer science; they are sequences of characters. It might be helpful to think of them as sentences, but not every sentence represents a string. We have seen how they work with our “Hello World” app. Strings are a separate data type with their own library. To manipulate and work with them a program needs an appropriate header – the string library. These libraries have methods and features that allow the program to manipulate strings.

Operation: String Concatenation

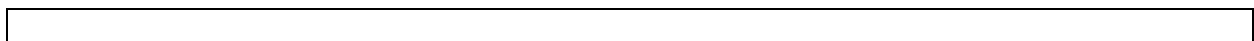
String concatenation is when different strings are put together into one string. It is combining strings. This is one of the most common functions of a program. For instance, when you enter your name at a website or a game and then you switch to another page and the screen greets you “Hello, [your_name]!”, the program has used concatenation to do that. It has combined “Hello,” with your name and “!” in the end.

In C++, string concatenations are performed through the “+” operator. Let’s see this in action by making a program that prints an imputed string with a greeting. For this to work, we must include the `<string>` library, which contains objects and statements needed for handling strings.

Open a fresh instance of the IDE we have been using and follow these steps:

1. In addition to
`#include <iostream>`, also include the `<string>` library
2. Omit `using namespace std;`
3. Remove the current filler code within `int main(){...}`
4. Declare and set the variables `firstName = "John"` and `lastName = Doe`
`fullName` by adding `firstName` and `lastName` with the “+” operator
5. Declare and calculate
6. Print out the result as “Hello John Doe.”

It should look something like this:



C++

Shortcuts

Reset

Copy

```
1 #include<iostream>
2 //Call the library for string handling
3 #include <string>
4 int main()
5 {
6     //Declare variables and complete
7     std::string firstName = "John ";
8     std::string lastName = "Doe";
9     std::string fullName = firstName
10
11     // prints results
12     std::cout<<"Hello "
13     << fullName <<"\n";
14
15     return 0;
16 }
17
```

Input
Does
Here...

Copy

➔ Run

➔ Run+URL (Generates i

Generated URL:

Copy

<https://ide.geeksforgeeks.org/8781-p190E>

Time(sec) : 0

Memory(MB) : 3.3327872821045

Output:

Copy

Hello John Doe.

String Concatenation

This screenshot is a successful test run of our string concatenation program. We encourage you to open up this program in another window and compare it to your program. You can access the successful run at this URL: <https://ide.geeksforgeeks.org/B7XErpI9OE>.

Note: Be sure to annotate your code. You can shorthand the instructions and use them as an impromptu algorithm to annotate your code. This will help you keep the program in order.

Usually, in forums, the concatenated text input is kept small as possible and the formatting is done by the program. In the example, we manually have to add space after “Hello” and “John”, but the program adds space. Now that you have seen how the “+” operator works, how might you add space? See if you can figure it out by playing with the code in your editor. If things fall apart you can always return to our successful IDE workspace by clicking on the URL.

Note:

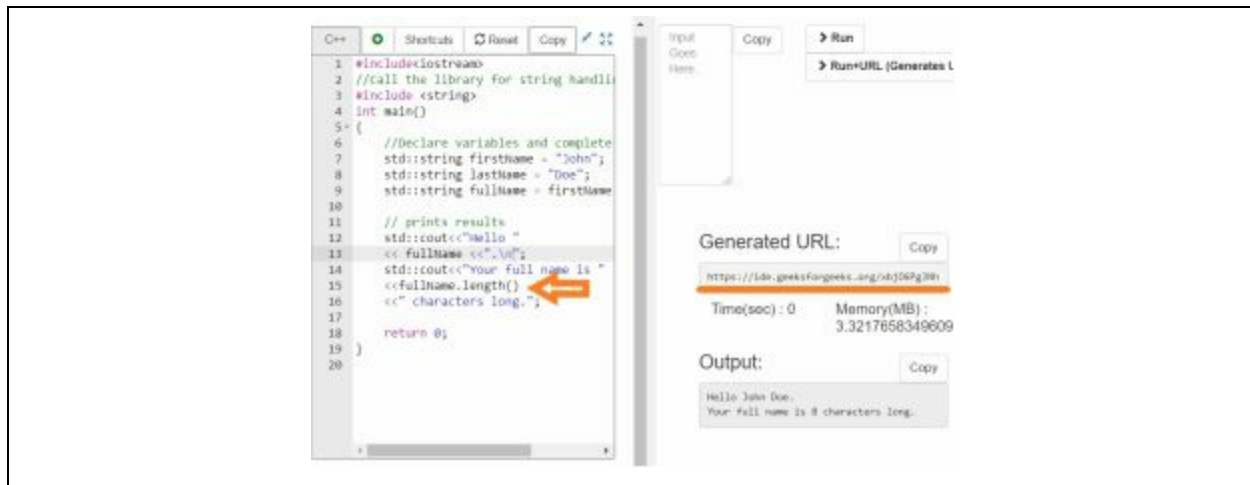
The “+” operator is used for both adding numbers and concatenating strings. Trying to use the operator to add a string and a number will produce an error. You can test this on your own using a modified program from the previous chapter. You can also check out a successful run of the program here that shows how to concatenate numbers as strings at this URL: <https://ide.geeksforgeeks.org/gIPmslDwF7>. Uncomment the declarations to try to add numbers and strings to see the error for yourself. The error is very long and verbose, as it conflicts across the `<iostream>` and `<string>` libraries.

String Objects: Length() Attribute

We know from real-world experience that strings, like many things, can come in different sizes. This is unlike other data types we have explored so far. C++ makes this possible because it construes strings as objects with attributes. C++ defines a string length under the *Length()* attribute as you would expect in an object. To access length, the string must be attached to a

variable and you must use the “.” syntax. It should look something like this:
`[string_name].length()`

Let’s return to our program and retrieve the length of *fullName*. We will need to modify our code by adding a new line with the syntax above and another *cout*, like this:



```
1 #include <iostream>
2 //Call the library for string handling
3 #include <string>
4 int main()
5 {
6     //Declare variables and complete
7     std::string firstName = "John";
8     std::string lastName = "Doe";
9     std::string fullName = firstName
10
11     // prints results
12     std::cout<<"Hello "
13     << fullName <<".\n";
14     std::cout<<"Your full name is "
15     <<fullName.length()
16     <<" characters long.";
17
18     return 0;
19 }
20
```

Generated URL: <https://ide.geeksforgeeks.org/xb56Pg3Nh>

Time(sec): 0 Memory(MB): 3.3217658349609

Output: Hello John Doe.
Your full name is 8 characters long.

Print `fullName.Length()`

This screenshot is a successful test run of our modified string concatenation program. We encourage you to modify the program at this URL: <https://ide.geeksforgeeks.org/B7XErpI9OE>.

Once you have modified the program, you can compare it to this successful test run by opening it in another window. You can access this saved IDE window at the following URL: <https://ide.geeksforgeeks.org/xbjO6Pg3Nh>.

String Indexes and Arrays

Recall our conversation on objects. Vector arrays are also a special mathematical phenomenon with their own special attributes. An **array** is an orderly arrangement of objects – often rows, columns, or a matrix – where each has its own specific location. Strings are stored in arrays made up of characters. Therefore, a string array is orderly arrangements of characters where each character in a string is indexed in its own location. Arrays are accessed by using `[]` to find a character or any other items in an array. String indexes start with `[0]`. Let’s see this in action by changing our program so that it prints out the 3rd character in *fullName*.

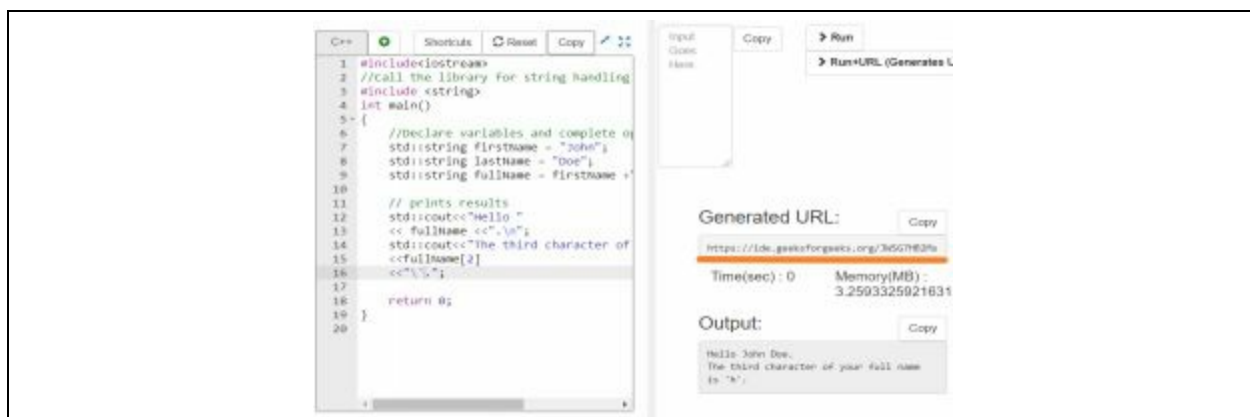
Note:

Remember that the index starts with [0]. Therefore the 3rd character in the string array would be at [2], not 3.

We are going to use this syntax to do so:

`fullName[x]`

The *x* stands for the index of the character we would like to print out. Remember to also edit the cout to properly print the letter. You will also need to use escape sequences like this:



The screenshot shows a C++ program in an IDE. The code defines two strings, 'firstName' and 'lastName', and concatenates them into 'fullName'. It then prints 'Hello ' followed by the full name, and the third character of the full name (index 2). The output shows 'Hello John Doe.' and 'The third character of your full name is: 'o'.'

```
1 #include<iostream>
2 //call the library for string handling
3 #include <string>
4 int main()
5 {
6     //declare variables and complete of
7     std::string firstName = "John";
8     std::string lastName = "Doe";
9     std::string fullName = firstName + " " + lastName;
10
11     // prints results
12     std::cout<<"Hello "
13     << fullName <<"\n";
14     std::cout<<"The third character of
15     <<fullName[2]
16     <<"\n";
17
18     return 0;
19 }
20
```

Generated URL: <https://ide.geeksforgeeks.org/3G57H83H>

Time(sec): 0 Memory(MB): 3.2503325921631

Output:

```
Hello John Doe.
The third character of your full name
is: 'o'.
```

Access fullName String Array

This screenshot is a successful test run of our modified string attributes program. We encourage you to modify the program at this URL: <https://ide.geeksforgeeks.org/xbjO6Pg3Nh>.

Once you have modified the program, you can compare it to this successful test run by opening it in another window. You can access this saved IDE window at the following URL: <https://ide.geeksforgeeks.org/JWSG7HB2Ms>.

In our examples we have relied a lot on *cout* capabilities, but, as I have alluded to, there are other strings handling capabilities used by programmers to field content from different sources. These are *cin* objects and the *getline()* function.

Cin and getline() function

In examples from the previous chapter, we have used *cin*. Like *cout*, *cin* is an object in the `<iostream>` library and it allows a program to receive input using the “>>” operator. Like *cout*, *cin* objects must be defined with the `std` data type. Let’s modify our concatenation programs so that it receives the input of “John Doe”. You can use your name if you like, as it can be fun to do so. You will need to follow these instructions:

1. Navigate to the string concatenator program at the following URL:

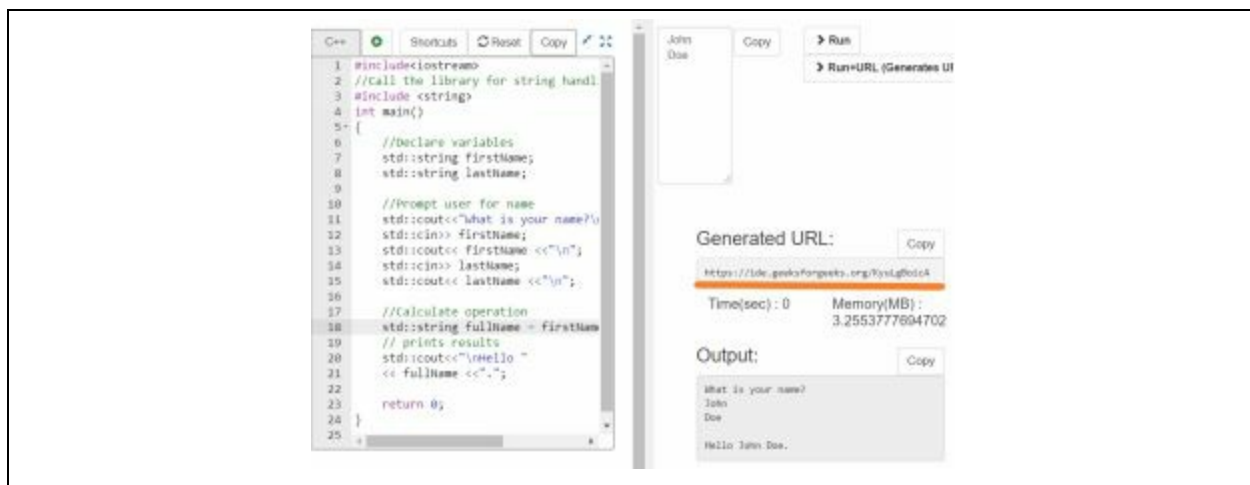
<https://ide.geeksforgeeks.org/B7XErpI9OE>

2. Declare the variables
3. Prompt the user for their name
4. Calculate the operation by using the following code snippet:

...fullName = firstName + " " + lastName...

5. Print the results

If you find yourself stuck, here is how it should look:



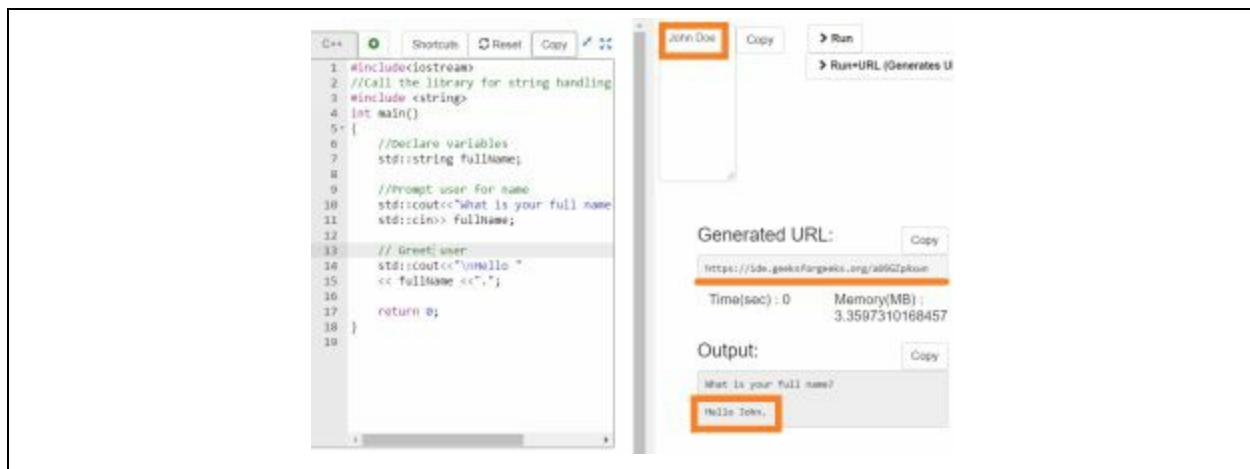
String Concatenation with cin

This screenshot is a successful test run of our modified string concatenation program using *cin*. We encourage you to open up this program in another window and compare it to your program. You can access the successful run at this URL: <https://ide.geeksforgeeks.org/KyuLgBoicA>.

Note: Be sure to annotate your code. You can shorthand the instructions and use them as an impromptu algorithm to annotate your code. This will help you keep the program in order.

The program will work only when the user enters their first and last name separately. But most peoples 'names are stored or put in as one single string. Let's change our program so that it greets the user when they enter their full name. You will need to follow these steps:

1. If not there already, or you need to reset the workspace, go to the previous workspace at the following URL:
<https://ide.geeksforgeeks.org/KyuLgBoicA>
2. Declare only the *fullName* variable
3. Prompt the user for their full name using *cin*
4. Greet the user using *cout* and the input *fullName*



String Termination with *cin*

This screenshot is an unsuccessful test run of our modified string concatenation program using *cin*. While the program completed with no errors, we did not get an expected result. If your program had errors after your modifications or did not get a similar result, we encourage you to use our successful run to troubleshoot your workspace. Open up this successful run in another window and compare it to your program. You can access the successful run at this URL: <https://ide.geeksforgeeks.org/aU6GZpAxwn>.

You have witnessed typical *cin* behavior. When *cin* encounters a space it reads it as a string terminator and stops. This is why “Doe” is not added to the output. To get it to read the entire string with spaces, we must use the *getline()* function from the `<string>` library; it extracts all characters from the input and turns them into a string array.

Note:

getline() is usually preferred over *cin* to ensure that programs can read fielded text from secondary sources, particularly since *getline()* can read and aggregate different sorts of characters without errors.

Here is an example of *getline()* functions used well:
<https://www.geeksforgeeks.org/getline-string-c/>

In this chapter, we have done quite a bit. We have looked at specialized functions, accessing objects, text manipulators, and operators. We will cover them in more detail in the coming chapters. I hope you have seen the role of operators in directing data and lending program functionality. In the following chapter, we are going to be looking at operators and how they give programs decision-making capabilities.

Chapter 4: Operations in C++

We talked about how C++ is mainly a back-end, server-side language that is suited for controlling servers and other decision-making utilities for delivering content. In the last chapter, we looked at small ways that C++ handles data by looking at strings. In it, we saw the central role that operations play. Operations allow us to control loops and decision making in programs. A better understanding of operations allows us to add more functionality to our programs. **Functionality** is how programmers use methods and features of a language to achieve an end. We have seen how these string handling methods work, but what would be better is if we used those features in a context that allows us to achieve a particular end, like content management.

Note: “Operand” describes the value or variable being operated on. “Operator” describes the symbol indicating the sort of operation being carried out (“C++ Operators,” n.d.).

The following is a list of operators in C++ (“Operators-In-C.png (800×533),” n.d.):

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

This table lists the operator symbols and categorizes them by (1) how many operands they accept at a time > (2) the type of operation they implement. For example, most arithmetic operators operate on two operands at a time.

All operators are characterized by how many operands they can operate on at a time and the types of operations they can execute. They are divided into three types:

Unary - operators that only accept 1 operand at a time

Binary - operators that only accept 2 operands at a time

Ternary - operators that accept more than 2 operands, or conditional operators with several arguments

They are further divided into 6 more types:

- Arithmetic
- Relational
- Logical
- Bitwise

- Assignment operators
- Other operators such as a conditional, address, and redirection

In this chapter, we will look at 5 types of operators and their functionalities.

Note: Operators have precedence, just as in mathematical equations. For example, parentheses, which are used to call a function, have higher precedence than addition or subtraction. However, you should try to keep your lines of code as simple as possible. This makes your programs easier to edit and debug. Having clean, uncomplicated code avoids bugs.

Binary Operators

Binary operators are the majority of operands you find in C++. They include 5 of the 6 types of operators. This makes them the perfect place to start. As the name suggests, binary operators compare two operands. They are perfect for decision-making and controlling switches.

Arithmetic Operators

They are the most common and well understood of all operators because of how much we use them even outside programming, since they are primarily mathematical. We use them to perform mathematical operations.

Note: In the string concatenation example, the “+” used to concatenate *was not an arithmetic function*. Instead, it was a function of the `<string>` library that handled the concatenation.

Most operators in this category require two operands. The only exceptions are the unary “++” and “--” operators which we use to increase or decrease within a loop. The binary operators are (Jaggi, 2015a):

“+”: the addition operator that adds two operands

“-”: the subtraction operator that subtracts one operand from another

“*”: the multiplication operator that multiplies two operands

“/”: the division “forward slash” operator that divides the first operand into the second

“%”: the modulus operator that returns the remainder from dividing the first operand into the second

Let’s make a *cout* program that demonstrates all of these 5 arithmetic operands. Create a new C++ workspace in the IDE and follow these instructions:

1. Declare two variables *a = 25* and *b = 3* as integers
2. Declare a variable, *result*
3. Print the variables *a* and *b*

4. Add a and b and print the result
5. Subtract a and b and print the result
6. Multiply a and b and print the result
7. Divide a and b and print the result
8. Calculate the remainder from dividing a and b and print the result

Here is how that looks.

The screenshot shows a C++ program in an IDE. The code declares two integers, a and b , with values 25 and 3 respectively. It then performs four arithmetic operations: addition, subtraction, multiplication, and division, printing the results for each. The output window shows the following results:

```

a+b is: 28
a-b is: 22
a*b is: 75
a/b is: 8
a%b is: 1

```

Arithmetic Routine Program

This screenshot is a successful run of the arithmetic routine program displaying all the arithmetic functions. You can access a copy of the raw code in the index.

If you get errors, I encourage you to use our successful run to troubleshoot your workspace. Open it up to compare the code. Here is where to find it:

<https://ide.geeksforgeeks.org/vDcCH7f5EQ>

Relational Operators

Most programs, not just in C++, use conditionals and loops to control flow. Operators in this section are used to perform conditional switching. For instance, in games, the game engine adapts to how the player interacts with the game. It gives the gamer more of an engaging and dynamic experience.

If the gamer completes challenges quickly, it might mean they find the game

too easy. As a result, the game adjusts by increasing difficulty, but not too much. If the game is too hard, the engine might respond to that by making it a little easier to play. This is called dynamic gameplay, and it happens because coding languages have conditional switching. Conditional switching, the adaptation, happens through relational operators that compare two values. All relational operators are binary (Jaggi, 2015b). Here they are:

“`==`”: “Equal” operator that checks whether two given operands are equal. If the operands are equal, it returns a Boolean “true”; if not, it returns false.

“`!=`”: “Not-Equal” operator that also checks whether two given operands are equal or not. However, this operator returns a Boolean true if the operands are not equal and returns false if the operands are equal.

“`>`”: “Greater-than” operator that checks if the operand on the left is greater than the operand on the right. If so, this operator returns a Boolean true. Otherwise, it returns false.

“`<`”: “Less-than” operator that checks if the operand on the left is less than the operand on the right. If so, this operator returns a Boolean true. It returns Boolean false otherwise.

“`>=`”: “Greater-than or equal-to” operator that checks if the operand on the left is greater than or equal to the operand on the right. If so, this operand returns a Boolean true. If not, it returns a Boolean false.

“`<=`”: “Lesser-than or equal-to” operator that checks whether the operand on the left is less than or equal to the operand on the right. If so, this operand returns a Boolean true. If not, it returns false.

Like we did with the arithmetic operators, let’s create a *cout* program which demonstrates relational operators. We are going to be using conditional *if()* statements in our code. An *if()* statement executes the code within its curly braces when a specified condition is met. If the condition is not met it will execute the code in the *else{}* statement. The *else* does not have a condition to check; it executes when the specified condition in *if()* is not met. Conditional statements always use a relational operator to test a condition.

Note:

The `if()` and `else` statement is case sensitive. Both “if” and “else” must be lowercase. Additionally, while `if()` can be executed without an `else`, the missing `else` statement can cause an error if the condition is false. `else` statements also cannot be implemented without an `if()` statement. We will describe `if()...else` statements in more detail in later chapters

To create our program we have to follow the following steps.

1. Open and edit the arithmetic routine program using this URL:
 - a. <https://ide.geeksforgeeks.org/vDcCH7f5EQ>
2. Use the same variables from the arithmetic routine. Be sure to clean the code of any unneeded variables.
3. Print an explanation of the program
4. Test `if(a > b)` and print the result
 - a. Create an `if` statement printing out the result if greater than
 - b. Create a nested `else` statement printing out the result if false

Note: For each `if()...else` you will have to create `acout` for all possibilities. For our simple program, it is only two possibilities for each statement.

5. Test `if(a >= b)` and print the result
6. Test `if(a < b)` and print the result
7. Test `if(a <= b)` and print the result
8. Test `if(a == b)` and print the result
9. Test `if(a != b)` and print the result



Relational Routine Program

This screenshot is a successful run of the relational routine program displaying all the relational functions.

10.

If you did not get similar results, go look at our code here: <https://ide.geeksforgeeks.org/dFD3cEk85y>. Compare and figure out where you went wrong. An exercise like this one will equip you with debugging skills.

Also, take your time to appreciate the code and understand it.

Logical Operators

Logical operators add more complexity to relational operators. These are comparable to human ideas of “and” or “or”. They also have qualities that switch the meaning of the relational operator. **Logical operators** are used in conjunction with relational operators to combine two or more conditions to describe a constraint. They make us catch a lot more and do a lot more with our conditionals. There are three logical operators (Jaggi, 2015b):

“&”: the logical “AND” operator returns a Boolean true when both the conditions are met and false if they aren’t.

“||”: the logical “OR” operator returns a Boolean true when one (or both) of the conditions are met, and false if none of the conditions are met.

“!”: the logical “NOT” operator returns a Boolean true if the conditions in consideration are not satisfied. If one of the conditions is true, then it returns a Boolean false.

Logical operators are used a lot in machines. Imagine a “NOT” operator in the security system of a bank. Let’s say locked doors are “0”[false], and unlocked doors are “1”. Because the bank should stay secure, our default value is “0.” This is called a failed closed system, because if things go wrong the doors will be closed (the default). All signals that go to the door are controlled by a “NOT” signal: If nothing is going on, the doors should remain locked, and if there is robbery they should also remain locked. Doors that open during a robbery will get a “NOT” signal to fail-close. But in a fire, these doors should open, so the fail-closed “NOT” signal will return true to open doors.

Let’s explore logical operators by building a logical routine program. Follow these instructions:

1. Open and edit the relational routine program using this URL:

<https://ide.geeksforgeeks.org/dFD3cEk85y>

2. Edit these additional variables to test the logical operators:

`a = 5, b = 0, c = 12, d = 24`

3. Print an explanation of the program
4. Test if(`a > b` && `c == d`) and print the result
 - a. Create an if statement printing out the result if greater than
 - b. Create a nested else statement printing out the result if false
5. Test (`a > b` || `c == d`) and print the result
6. Test (`!b`) and print the result
7. Clean the code of any unnecessary lines

Here is how it looks:



The screenshot shows a C++ IDE with a code editor on the left and a console/output window on the right. The code in the editor is as follows:

```
7 // print an explanation of the program
8 // print a and b
9 // print a and b
10 // print a and b
11 // print a and b
12 // print a and b
13
14 // logical AND example
15 if (a && b) {
16     cout << "a is greater" << endl;
17 }
18
19 // logical OR example
20 if (a || b) {
21     cout << "a is greater" << endl;
22 }
23
24 // logical NOT example
25 if (!b) {
26     cout << "b is zero" << endl;
27 }
28
29 // logical AND example
30 if (a && b) {
31     cout << "a is greater" << endl;
32 }
```

The output window on the right shows the following text:

```
Generated URL: https://ide.geeksforgeeks.org/x9ixjUYDBG
Output:
This program will go through a
logical routine with the following
variables: a is 5 and b is 0
a is 5 and b is 0
AND condition not satisfied--
(a && b) is 0
a is greater than b OR b is equal
to 0 is true
```

Logical Routine Program

This screenshot is a successful run of the logical routine program displaying examples of all the logical functions. You can access this workspace with this URL: <https://ide.geeksforgeeks.org/x9ixjUYDBG>

As you can see, the “AND” function may be difficult to print with just one `if()...else` statement when producing a false state. Either relational expression can be false to produce a false AND. However, the program can be fixed to produce an exact answer by nesting another `if()...else` statements. Modify this exercise to create a program that can explain itself better!

Hint: You only need one additional `if()...else` statement. You can check your work against ours with this URL -- <https://ide.geeksforgeeks.org/nh8yYV9Kjy>.

Let’s take a break from routines and explore other ways we can use logical operators. Logical operators are based on machine logic, especially logical gates. Logical gates are electronic circuits that have one or more inputs while having one output. Logical gates have the same function as logical operators. “AND” gates only pass “1” when all inputs are “1”; “OR” gates pass “1” when they receive a “1” from any of their inputs; “NOT” passes “1” when they receive a “0” as their input.

Logical gates can control the flow of data in a circuit, the same as our logical operations when combined with a conditional statement. It is called a **conditional short-circuit** in logical operators.

Logical AND operators and OR operators short-circuit. Recall how logical AND returns a true Boolean value when both relational operators are conditionally true. What’s more, relational operators are assessed in the order that they appear in a conditional statement – if the first one is false the second

one will not be evaluated, so it short circuits.

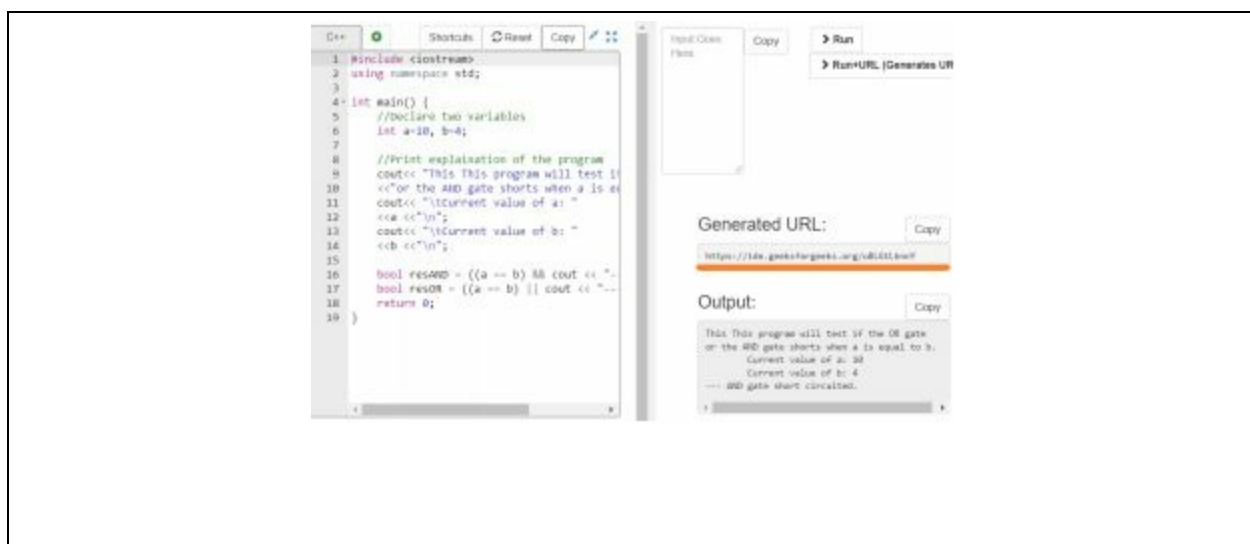
OR operators do this differently. Only one relational operator has to return true for it passes a true Boolean value. So if the first relational operator returns true, it won't check the next one.

This is very useful because we can give more complex instructions to a program about how to behave in what circumstances. OR and AND are short-circuit opposites. Let's use this to write a program that tells when a condition short-circuits AND and OR gates.

Follow these instructions:

1. Open a new IDE workspace
2. Declare two *int* variables:
 $a = 10, b = 4$
3. Print an explanation of the program
 - a. This program will test if the OR gate or the AND gate shorts when a is equal to b.
 - b. Print current values of a and b
4. Declare and calculate two *bool* variables:
5. `resAND = ((a == b) && std::cout << "OR circuit shorted")`
6. `resOR = ((a == b) || std::cout << "AND circuit shorted")`

This is how it looks:



```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     //Declare two variables
6     int a=10, b=4;
7
8     //Print explanation of the program
9     cout << "This This program will test if
10    <<" or the AND gate shorts when a is equal to b.
11    <<" << endl;
12    cout << "Current value of a: "
13    << a << endl;
14    cout << "Current value of b: "
15    << b << endl;
16
17    bool resAND = ((a == b) && cout << "OR circuit shorted");
18    bool resOR = ((a == b) || cout << "AND circuit shorted");
19    return 0;
20 }
```

Generated URL: <https://ide.geekstargets.org/vd631bwl>

Output:

```
This This program will test if the OR gate
or the AND gate shorts when a is equal to b.
Current value of a: 10
Current value of b: 4
--- AND gate short circuited.
```

Short Circuit Detection Program

This screenshot is a successful run of our short circuit detection program. Using our knowledge of logical AND, logical OR, and their relationship to each other, we were able to develop a way to detect short circuits. You can access this workspace with this URL: <https://ide.geeksforgeeks.org/uBLGtLbnnY>

Try different values of *a* and *b* to see if you can trip the different “circuits.”

Consider a short circuit detection program. How would we use it in the real world? I have alluded to systems like these playing a role in bank security. The question is how it would achieve those things? You can tell your program that if a condition is not (“NOT”) met to refuse access, like if the password does not match.

An understanding of logical operators gives a good idea of how machines and systems make their decisions. Relational and logical operators give us more control over what happens in the program and how it relates to everything around it. We do this through various libraries, statements, and objects. So far we have written small programs, but in bigger programs using objective control like these can cause overhead for the compiler. That has a noticeable effect in programming environments and industries like gaming engines, content delivery, and others. Think about the servers at YouTube that have to deliver gigabytes of data per second and gaming engines that need to render in real-time: they have to reduce the amount of overhead while being able to utilize these control features. Thankfully, that can be achieved by using bitwise operators.

Bitwise Operators

On the circuit level, machines communicate by using AND and OR gates. It

gives them decision-making capabilities at a single bit data level. **Bitwise operators** use those microscopic level calculations to access computing processes at the machine level, bypassing objective programming. In other words, they are logical operators operating on numbers at a binary level.

To use them, you will need to know about binary. Humans conceptualize numbers using **base-10** numbers, also known as the decimal system. For instance, a number like 543 is understood as five-hundreds, three tens, and 4 single units. In base 10 it looks like this:

$$5 * 10^2 + 3 * 10^1 + 4 * 10^0 = 534$$

In binary the same number would look like this 0b1000010110 – the “0b” is just for us humans to understand it is a binary number. Binary numbers are **base-2**, they just look different. All binary numbers are presented as a stream of bits, which can either be in two states: 1 or 0. Like in base-10, the position of the number determines its value. The 534 example shows a binary number that is 10 bits long. Looking closely at the 534 streams we can see that the 2nd, 3rd, 4th, and 10th bits are on (moving right to left), and all the other bits are off (1 is on, 0 is off). In base 2 we calculate this as:

$$2^9 + 2^4 + 2^2 + 2^1 = 534$$

It is not easy for us humans to calculate it, but this is how computers communicate, in streams of data bits. They are more adept at recognizing and counting these numbers just as easily as if they were base-10 numbers to humans. Therefore, using bitwise operators that function at the computer’s natural level improves speed and performance ([Killian, 2012](#)) because it skips the translation of objective programming into machine language. This is why you will find bitwise operations are often used in competitive programming. Below is a list of C++ bitwise operators (“Bitwise Operators in C/C++,” 2014):

“&”: Bitwise AND takes two numbers (operands) and runs AND on every bit within the stream of those numbers. The results of the AND stream will be 1 if both are 1.

“|”: Bitwise OR takes two numbers(operands) and runs OR on every bit within the stream of those numbers. The results of the OR stream is 1 if any of the two is 1.

“ \wedge ” Bitwise XOR that takes two numbers as operands and does XOR on every bit within the stream of the two numbers. The result stream of XOR is 1 if the two bits are different.

“ \ll ” Left shift operator takes two numbers and left shifts the bits of the first operand. The second operand is used to determine the number of places to shift.

“ \gg ” Right shift operator takes two numbers and right shifts the bits of the first operand. The second operand is used to determine the number of places to shift.

“ \sim ” Bitwise NOT operator that takes one number and switches the state of all the bits (1s to 0s, 0s to 1s).

Note: Machines use registers and buffers to store the bits for calculation. A **register** is just an array of storage, where each bit gets its own place in the array. If you think of these data streams as an array, it will help you rationalize many of the bitwise operators, particularly the shift operators.

Let's write a bitwise routine program. First, we will need to create a bit register and study bit streams. This requires the `bitset<x>()` function for the `<bitset>` library. It allows you to print a binary stream with a length of `<x>`. Follow the instructions:

1. Open a new IDE C++ workspace
2. In addition to `<iostream>`, be sure to call the `<bitset>` library as well
3. Declare the operand variables and their values

```
unsignedinta = 60;//60 = 00111100
```

```
unsignedintb = 13;//13 = 00001101
```

4. Declare register c for the calculations

```
intc = 0;
```

5. Print operand values and their registers:

```
cout << "Operandregistersa:" << a
```

```
<< "andb:" << b
```

```
<< "\n\tregistera: - 0b" << bitset < 8 > (a)
```

```
<< "\n\tregisterb: - 0b" << bitset < 8 > (b) << endl;
```

Note: Since all numbers are less than 255, we will then use a bit stream of `< 8 >` for the entire exercise.

6. Calculate Bitwise AND for *a* and *b* in register *c*

```
c = ab;//12 = 00001100
```

7. Print the result and the register in *c*

Note: Since we are using bitwise functions, it is better to use `endl;` to terminate our lines. This is because this function clears the buffer and prevents overflow. This makes the program a little slower, but it ensures that our registers are clear. The speed difference is made up by using bitwise functions.

8. Calculate Bitwise OR for *a* and *b* in register *c*

```
c = a|b;//61 = 00111101
```

9. Print the resulting value and the register in *c*

10. Calculate Bitwise XOR for *a* and *b* in register *c*

```
c = a^b;//49 = 00110001
```

11. Print the resulting value and the register in *c*

12. Calculate Bitwise NOT for *a* in register *c*

```
c = ~a;// - 61 = 11000011
```

13. Print the resulting value and the register in *c*

14. Print a comparison register for *a* in register *c*

This will make it easier to compare with the shifting bitwise operations

15. Calculate Bitwise Left Shift for *a* << 2 in register *c*

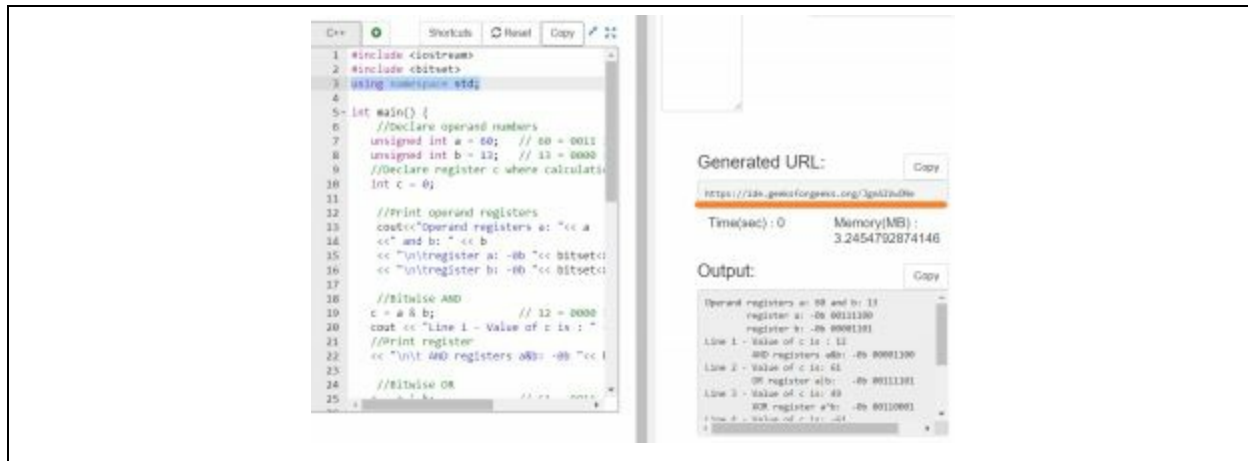
```
c = a << 2;//240 = 11110000
```

16. Print the result and the register in *c*

17. Calculate Bitwise Left Shift for *a* >> 2 in register *c*

```
c = a >> 2;//15 = 00001111
```


18. Print the result and the register in `C`



The screenshot displays a C++ IDE with a code editor on the left and a console/output window on the right. The code in the editor is as follows:

```
1 #include <iostream>
2 #include <bitset>
3 using namespace std;
4
5 int main() {
6     //Declare operand numbers
7     unsigned int a = 60; // 60 = 0011
8     unsigned int b = 13; // 13 = 0000
9     //Declare register c where calculation will be done
10    int c = 0;
11
12    //Print operand registers
13    cout << "Operand registers a: " << a
14    << " and b: " << b
15    << "\n\tregister a: -0b " << bitset<>
16    << "\n\tregister b: -0b " << bitset<>
17
18    //Bitwise AND
19    c = a & b; // 12 = 0000
20    cout << "Line 1 - Value of c is : "
21    << "\n\tAND registers a&b: -0b " << bitset<>
22
23    //Bitwise OR
24    c = a | b; // 61 = 0011
25    cout << "Line 2 - Value of c is : "
26    << "\n\tOR registers a|b: -0b " << bitset<>
27
28    //Bitwise XOR
29    c = a ^ b; // 47 = 0011
30    cout << "Line 3 - Value of c is : "
31    << "\n\tXOR registers a^b: -0b " << bitset<>
32
33    //Bitwise NOT
34    c = ~a; // -61 = 1111
35    cout << "Line 4 - Value of c is : "
36    << "\n\tNOT register ~a: -0b " << bitset<>
37
38    //Bitwise Left Shift
39    c = a << 2; // 240 = 1111
40    cout << "Line 5 - Value of c is : "
41    << "\n\tLeft Shift register a << 2: -0b " << bitset<>
42
43    //Bitwise Right Shift
44    c = a >> 2; // 15 = 0000
45    cout << "Line 6 - Value of c is : "
46    << "\n\tRight Shift register a >> 2: -0b " << bitset<>
47
48    return 0;
49 }
```

The output window on the right shows the following information:

Generated URL: <https://ide.geeksforgeeks.org/JgnAlVwONE>

Time(sec): 0 Memory(MB): 3.2454792874146

Output:

```
Operand registers a: 60 and b: 13
register a: -0b 00111000
register b: -0b 00001101
Line 1 - Value of c is : 12
AND registers a&b: -0b 00001000
Line 2 - Value of c is: 61
OR register a|b: -0b 00111101
Line 3 - Value of c is: 47
XOR register a^b: -0b 00110001
Line 4 - Value of c is: -61
Line 5 - Value of c is: 240
Line 6 - Value of c is: 15
```

Bitwise Routine with Register Printing

This screenshot is a successful run of our bitwise routine program. The output features visuals of the `C` register array. This will allow you to see how each bitwise is calculated and how it changes the register. This is a long program with a very long output. To study it fully, you can access it on our saved workspace using this URL: <https://ide.geeksforgeeks.org/JgnAlVwONE>

Note:

This program uses `using namespace std;` to simplify the program. It would be good practice to omit the namespace and declare each `cout` and `bitset` object separately.

The last binary operators we will discuss are assignment operators. **Assignment operators** are used for assigning value and dynamically changing values. In principles of programming, we talked about programs needing to be as efficient as possible while using a few instructions as possible. Advanced assignments operators combine with equal assignments and other various operations to reduce lines of code. They are all **atomic operations**, meaning they allow programmers to manipulate stored values within variables, and reassign them. Below is a list of common assignment

operators (Prabhu, 2018):

“ = ”: Equal assignment operator assigns the value on the right to the variable on the left.

“

+= ”: Operator for atomic addition, used to combine the ‘ + ’ and ‘ =

’ operators. This operator *adds* the value of the variable on the left to the value on the right. Then it saves the result to the variable on the left. For example, $x += y$ would be a shorter way of writing $x = x + y$ without having to use a separate arithmetic operator.

“

-= ”: The atomic subtraction used to combine the ‘ - ’ and ‘ =

’ operators. This operator *subtracts* the value of the variable on the left from the value on the right. Then it saves the result to the variable on the left.

“

*= ”: Operator for the atomic multiplication, used to combine the ‘ * ’ and ‘ =

’ operators. This operator multiplies the value of the variable on the left with the value on the right and saves the result to the variable on the left.

“

÷ ”: The atomic division operator is a combination of ‘ / ’ and ‘ =

= ’ operators. This operator divides the value of the variable on the left by the value on the right and saves the result to the variable on the left.

For example, x

$÷ y$ would be a shorter way of writing $x =$

$\frac{x}{y}$

without having to use a separate arithmetic operator.

Note: There are also bitwise atomic assignment operations: “ = ”, “ | = ”, and “ ^ = ”.

’ We will focus on the arithmetic-atomic functions for now.

Let’s create an assignment operator routine program to see how these assignment operations work. Follow these steps:

1. Open a new IDE workspace
2. Declare and initialize our variable
`int a = 10;`
3. Print an explanation of the program and the starting value

Note: When making programs that print, always make them easy to read! Make sure your program can explain itself.

4. Atomically add 10 and print the result
5. Atomically subtract 10 from our variable and print the result
6. Atomically multiply 10 and print the result
7. Atomically divide our variable by 10 and print the result

Here is how it looks:



The screenshot shows a C++ IDE with a workspace. The code in the editor is as follows:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // Assigning value 10 to a
6     // using "-" operator
7     int a = 10;
8     cout<<"this program will atomically manipulate the value of a is " << a;
9
10    // Assigning value by adding 10 to a
11    // using "+" operator
12    a += 10;
13    cout<<"\n atomically + 10: the value of a is " << a;
14
15    // Assigning value by subtracting 10 from a
16    // using "-" operator
17    a -= 10;
18    cout<<"\n atomically - 10 from a: the value of a is " << a;
19
20    // Assigning value by multiplying 10 to a
21    // using "*" operator
22    a *= 10;
23    cout<<"\n atomically * 10: the value of a is " << a;
24
25 }
```

The IDE also shows the output of the program:

```
Generated URL: https://ide.geeksforgeeks.org/Fbc63b0u1E
Time(sec): 0 Memory(MB): 3.3108610757448
Output:
This program will atomically manipulate the value of a
The value of a is 10
Atomically + 10: the value of a is now 20
Atomically - 10 from a: the value of a is now 10
Atomically * 10: the value of a is now 100
Atomically dividing a by 10: the value of a is now 10
```

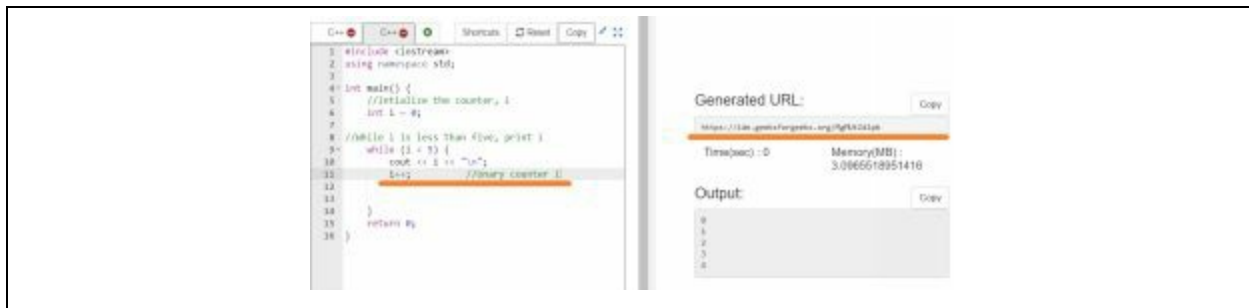
Assignment Routine

This screenshot is a successful run of our assignment routine program. This program has a sequential output, which means that each instruction impacts the result of the next instruction. This means if we were to change the order of the atomic operations applied to *a* that the output would be different. To study it fully, you can access it on our saved workspace using this URL: <https://ide.geeksforgeeks.org/Fbc63b0u1E>

These binary assignment operators combine assignments and arithmetic functions. They work well when controlling loops and conditional statements, too. Atomic assignment operators are closely related to unary arithmetic operators. Both are frequently used to control loops. Let's take a look at unary operations.

Unary Operations

Before we discuss unary operations, let's look at code that uses them.



While Loop featuring a Unary Operator

This screenshot is of a simple while loop that prints the value of the counter, i . The unary operation performs i

$= 1 + i$ for each iteration until i
 $= 5$ and the while loop ends. Omitting i
 $++$

will cause an error as the while loop would not be able to advance. We will talk about loops extensively in the decision making chapter. To study it fully, you can access it on our saved workspace using this URL:

<https://ide.geeksforgeeks.org/MgMUtZdlpb>

The $i = 1 + i$ is the same as $i++$. Using the unary form makes the code neater because it precludes the $+$ arithmetic operator and the $=$ assignment operator. We can also use an atomic addition operator to complete the code: $i+=1$ is the same as $i++$. But a format like that would require the compiler to store and operate on an additional operand. **Unary operators** are there to simplify code by only having one operand. Remember, unary operators are those operators that use one operand. Here are some of the other examples of unary operators (Kumar, 2017) :

“ $++$ ”: Increment operator used to increase the value of an integer by 1. It can be placed in front of a variable to increment the value immediately or after to temporarily save the value before increment.

“ $--$ ”: Decrement operator used to decrease the value of an integer by 1. Similar to the increment, programmers can also implement

pre-decrement and post-decrement instructions.

“ - ”: Unary minus operator that is used to change the sign of a variable or argument. Performing a unary minus operation on a negative integer will make it positive and vice versa.

“ ! ”: NOT operator is used to reverse the Boolean logical state of an operand. For example, if a variable *x* has a Boolean value of *false*, *!x* will be *true*.

“ & ”: Address of operator that is used to point to the address of a variable.

Let's make a unary operator routine to see how the pre- and post-instructions work. Follow these instructions:

1. Open a new IDE workspace
2. Declare and initialize our variable and a buffer
`int a = 10, buf`
3. Print an explanation of the program and the starting value
4. Calculate a post-increment equal to the buffer and print the result
5. Calculate a post-decrement equal to the buffer and print the result
8. Calculate a pre-increment equal to the buffer and print the result
9. Calculate a pre-decrement equal to the buffer and print the result



The screenshot displays a C++ IDE with a code editor on the left and an output window on the right. The code in the editor is as follows:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 10, buf;
6     cout << "This program will use unary operators to increment and decrement variable a and a buffer\n";
7     cout << "The value of a is " << a << endl;
8
9     // post-increment
10    buf = a++;
11    // a becomes 11 now
12    cout << "Post-Increment: a is " << a << endl;
13
14    // post-decrement example:
15    buf = a--;
16    // a becomes 10 now
17    cout << "Post-Decrement: a is " << a << endl;
18
19    // pre-increment example:
20    // buf is assigned 11 now since a is up
21    buf = ++a;
22    // a and res have same values = 11
23    cout << "Pre-Increment: a is " << a << endl;
24
25    // pre-decrement example:
26    // buf is assigned 10 now since a is down
27    buf = --a;
28    // a and res have same values = 10
29    cout << "Pre-Decrement: a is " << a << endl;
30}
```

The output window on the right shows the following text:

Generated URL: <https://ide.gatoforgeeks.org/ncv7ky04>

Time(sec): 0 Memory(MB): 3.2181273153687

Output:

```
This program will use unary operators to increment and decrement variable a and a buffer
The value of a is 10
Post-Increment: a is 11 and buf is 10
Post-Decrement: a is 10 and buf is 11
Pre-Increment: a is 11 and buf is 11
Pre-Decrement: a is 10 and buf is 10
```

Unary Operator Routine with Post- and Pre-Operations

This screenshot is of our unary operator routine program. To study it fully, you can access it on our saved workspace using this URL:

<https://ide.geeksforgeeks.org/mvcYtky6C4>

The program operates on a and while it calculates, it stores the value in a buffer we can observe. Comparing a with the buffer, the post-operation delays changing the value of a ; in contrast, pre-operations change the value immediately.

On top of all this, we have the `sizeof()` operator. It looks like a function but it is categorized as an operator. It checks the size of an object and returns the size in bytes.

Recall our conversation about bit streams. What I didn't say is that they are often divided into a group of 8. One byte is a single 8-bit stream that can present 255 numbers. In our bitwise routine, our registers were 1 byte long, that was because we were working with numbers that were less than 255.

Let's write a program that can get the size of an array using `sizeof()`. Here we go:

1. Open a new C++ IDE workspace
2. Declare and initialize our array

```
intarr[] = {1,2,3,4,7,98,0,12,35,99,14}
```

3. Print an explanation of the program and the array

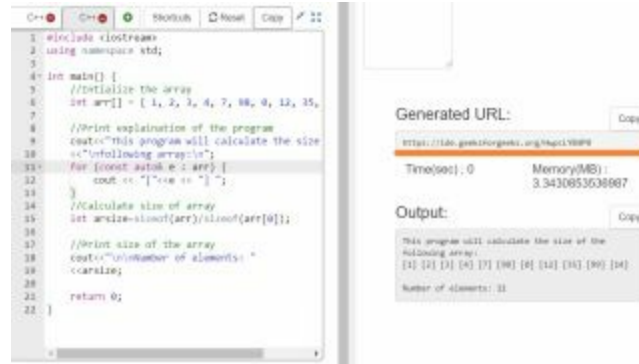
Note: To print the array you will have to create a for loop. We will discuss for loops in the next chapter. For now, just use these lines of code:

```
for(const auto& e: arr){
    cout << "[" << e << " ";
}
```

4. Calculate the size of the array

```
intarsize = sizeof(arr)/sizeof(arr[0]);
```

5. Print the size of the array



```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     //Initialize the array
6     int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22 };
7
8     //Print explanation of the program
9     cout << "This program will calculate the size of the following array:\n";
10
11     //Print the array
12     for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++) {
13         cout << arr[i] << " ";
14     }
15
16     //Calculate size of array
17     int arr_size = sizeof(arr)/sizeof(arr[0]);
18
19     //Print size of the array
20     cout << "Size of array: " << arr_size << endl;
21
22     return 0;
23 }
```

Generated URL: <https://ide.geeksforgeeks.org/HwpcLYB4P0> Copy

Time(sec): 0 Memory(MB): 3.3430853638887

Output: Copy

This program will calculate the size of the following array:
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22]
Size of array: 22

Array Size Reporting Program

This screenshot is of our array size reporting program. We encourage you to compare your result to ours. To study it fully, you can access it on our saved workspace using this URL: <https://ide.geeksforgeeks.org/HwpcLYB4P0>.

That was a good example of two unary operations: *sizeof()* and the address pointer *&*. The *for(){}* loop we used to print the array is called a copy constructor call. **Constructor copy calls** consist of a *for()* loop with an index variable, an *Addressof* operator, and a container object with an index. The copy constructor sets up *[0]: [the size of arr]* by using the *Addressof* pointer to automatically deduce the *[size of arr]* and print its contents. In simple words, a copy constructor uses *Addressof* to reduce the size of the program.

The *sizeof()* operator is an objective operation and it uses a single operand, the *arr* array object. They both have disadvantages and advantages. The *sizeof()* operator uses one operand and is easier to spot without esoteric memory registers. The call combination had to use an additional object (*e*), to print the array but it would have done with less incrementing operators. We are going to discuss loops and decision making features of C++ in more detail in the next chapter. For now, let's turn our attention to ternary operators.

Ternary Operators

C++ only has one ternary operator. **Ternary operators**, also called conditional operators, take 3 or more operands – they are Boolean logic-based operators. The result is always determined by whether or not the first expression is true, then the second will be evaluated; if the first expression is false, the third expression will be evaluated. If you recall our *if()...else* statement in the relational program routine, it had three parts: an initial expression that the function evaluates, an expression it evaluates if the initial expression is true, and an else statement which is evaluated when the initial expression is not true. Conditional operators have the same components as an *if()...else* statement. They are just more compact.

Let's make a program that illustrates these similarities:

1. Open a new IDE C++ workspace
2. To use the conditional operator you will have to call the `<bits/stdc++.h>` library
3. Print an explanation for the conditional operator

This program will pick the greatest one using two methods: a conditional operator and an *if()...else* statement

We expect a result of 5.

4. Declare the variable and execute the conditional operator


```
int arsize=sizeof(arr)/sizeof(arr[0]);
```

5. Print an explanation for the *if()...else*

```
if(2 > 5) then print 2
```

```
else then print 5
```





The screenshot shows a C++ IDE with a code editor on the left and a console/output window on the right. The code in the editor demonstrates conditional operations using the ternary operator and if-else statements. The output window shows the results of the program execution, including the generated URL, execution time, memory usage, and the program's output.

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     cout << "This program will pick the greatest
7     << "method is conditional operator and"
8     << "an 'if()... else statement.'"
9     << "We'll expect a result of 5."
10    cout << "Execute expression using"
11    << " ternary operator: ";
12    // execute expression using
13    // ternary operator
14    int a = 2 > 5 ? 2 : 5;
15    cout << a << endl;
16
17    cout << "Execute expression using "
18    << "if else statement: ";
19    // Execute expression using if else
20    if ( 2 > 5 )
21        cout << "2";
22    else
23        cout << "5";
24
25    return 0;
}
```

Generated URL: <https://ide.geeksforgeeks.org/9kJBAaQJtK> Copy

Time(sec): 0 Memory(MB): 3.222738259991

Output: Copy

This program will pick the greatest one using two methods: conditional operator and an if()... else statement. We expect a result of 5.

Execute expression using ternary operator: 5

Execute expression using if else statement: 5

Conditional Operations Demo

This screenshot is of our conditional operations demo. This program has no real practical functionality. However, for our purposes, it does demonstrate the similar structures between conditional operators and *if()...else* statements. To study it fully, we suggest that you access the copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/9kJBAaQJtK>.

In this chapter, we looked at 3 types of operators: unary, binary, and ternary. As you have seen, these operators allow programmers to apply powerful aspects of C++. For instance, binary operators allow machines to make logical decisions, compute values, and control processes by changing functions. Unary functions help us shrink and simplify complicated code. Ternary operators allow us to be more succinct in our code. They are all central to making decisions and switching in C++.

Chapter 5:

Decision Making in C++

We're going to talk about decision-making functions; these are functions like *if-else* loops. Programming rests a lot on decision making, and decision making is informed by logical expressions involving operators we have seen in chapter 4. Decision-making functions combined with logical expressions give machines the ability to make decisions based on criteria selected by programmers.

In our *sizeof()* exercise in Chapter 4 we used a *for()* loop in a copy constructor which fielded the array, printing all indices. The unary operator *Addressof* with an auto data type allowed us to implement the *for()* loop, without an additional counter for the loop. The simplicity of the copy constructor is not so obvious if you have never had the experience of initializing and implementing loops yourself.

Loops

Programmers spend a lot of time automating mundane, repetitive tasks through scripts and routines. What allows scripts and routines to achieve this are loops. As the name suggests, loops repeat the same instructions as long as a certain condition is met (“C++ While Loop,” n.d.). Loops are often used to navigate aggregated objects with indices like arrays and vectors. For instance, you might write a loop that keeps searching an array until a specific item is found. There are two types of loops: entry-controlled loops and exit loops.

In our unary arithmetic example, we had a *while()* loop with a counter. The counter is an arbitrary variable that is used with a unary operator to increment an indicated expression. It is an example of an entry controlled loop – it tests a condition before it executes code. Let’s modify that example into a *while()* loop to explore its components. We will use it to print out “Hello World” 10 times. Follow these instructions:

1. Open the following IDE C++ workspace:

<https://ide.geeksforgeeks.org/MgMUtZdlpb>

2. Print explanation of the program

This program will number and print 10 lines--

3. Change the conditional test to $i < 11$
4. Print “Hello World” 10 times with numbered lines



The screenshot displays a C++ IDE workspace. On the left, the code editor shows a program that initializes a counter `i` to 1 and enters a `while (i < 11)` loop. Inside the loop, it prints the line number and "Hello World", then increments `i`. On the right, the IDE interface shows the "Generated URL" as `https://ide.geeksforgeeks.org/0Kj0LTSht`, execution time as 0 seconds, memory usage as 3.2568101235962 MB, and the output as 10 numbered lines of "Hello World".

While Loop Demo

This screenshot is of our while loop demo. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/0WjOL6TSh4>.

This program shows us how incremental counters work. For it to work, the number must start with 1, not 0. Most people are used to numbers working that way. To do this, we initialized the counter at 1. Yet if we set the condition to 10 it will give us 9 print outs. To get 10 we have to increase the number to 11.

Another example of a controlled loop is the `for()` loop function; it requires three expressions as inputs (Agarwal, 2017a):

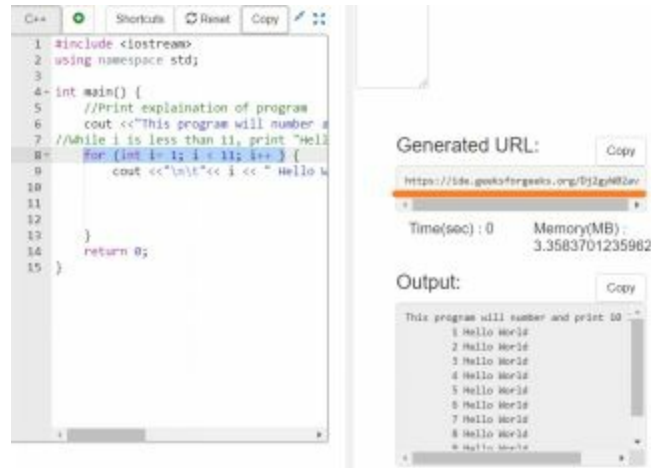
1. An initialization expression that declares a data type and a variable. This variable can be either a counter or an indexing variable.
2. A test expression that produces a Boolean *true or false*. If the condition is true then the for loop will continue to loop its specified block of code.
3. An update expression that increments the counter or the indexing variable

Let's see how they work by writing a program similar to our *while()* loop example. Follow these instructions:

1. Open the following IDE C++ workspace:
<https://ide.geeksforgeeks.org/0WjOL6TSh4>
2. Print explanation of the program
This program will number and print 10 lines of Hello World
3. Implement the `for()` loop

```
for(int i = 1; i < 11; i++)
```

4. Print "Hello World" 10 times with numbered lines
5. Clean the data of any unneeded code and comments



```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     //Print explanation of program
6     cout << "This program will number and print 10 'Hello World's" << endl;
7     //While i is less than 11, print "Hello World"
8     for (int i = 1; i <= 11; i++) {
9         cout << "\n\t" << i << " Hello World" << endl;
10    }
11    return 0;
12 }
```

Generated URL: <https://ide.geeksforgeeks.org/Dj2gyW02av>

Time(sec): 0 Memory(MB): 3.3583701235982

Output:

```
This program will number and print 10 "Hello World's
1 Hello World
2 Hello World
3 Hello World
4 Hello World
5 Hello World
6 Hello World
7 Hello World
8 Hello World
9 Hello World
10 Hello World
```

For Loop Demo

This screenshot is of our for loop demo. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/Dj2gyW02av>.

Note: Make sure you clean your code of unnecessary lines and update your comments. While this does not impact the performance of your code, it can make things confusing for other programmers on your team that have to implement your code. Do you see a mistake here?

Now that we see how *for()* loops work and how they are constructed, let's look at the copy constructor we used earlier. The expression used for the copy constructor call was:

for(const auto& e: arr)

“E” being the indexing variable and “arr” the variable array. We saw that the *for()* loop needs three expressions to work: the initialization, the test expression, and the update expression.

The copy constructor has an initializing function (*const auto& e*). When you look closely, you can see this expression also works as a test expression, because the & is an Addressof operator that points to *e* which stands for indices in the *arr* array. The : is an operator to break into class. It allows the attributes of the container *arr* array to pass to the *e* variable. The *for()* loop will print the *e* until the end of the array is reached.

The copy constructor call is good for a situation where the size of the container is unknown and you want all items printed out. *For()* loops define the number of iterations that have to take place from the get-go, while copy

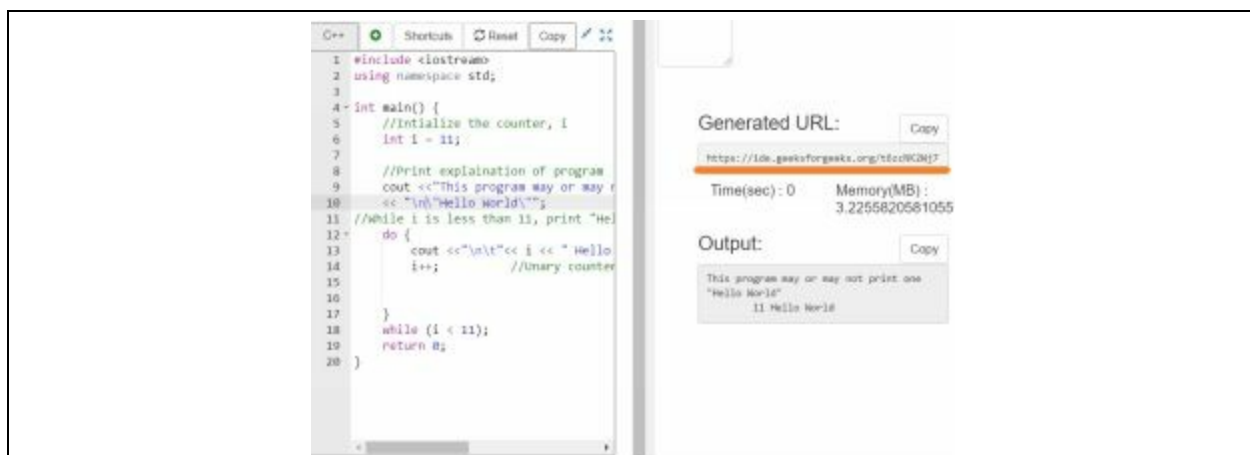
constructors calls are defined by the size of the container. *While()* loops are best for the situation where the condition can change at any time, during which you want the iterations to stop.

The *do{}..while()* loop is for situations where the *while()* code may impact the test. *Do{}..while()* loops are exit controlled, meaning code is executed until a condition is met. The code is executed at least once before checking the test condition. Think back to that bank security example. The door will always fail-close, receiving a NOT signal, except when there is a fire. Here is how a *do{}..while* loop will look for that:

```
do{Keepthedoorslocked}  
while(fire = false);
```

The door will remain closed as long as fire is not detected. *Do{}..while()* loops are useful for many fail-close systems. Other applications include data correction in servers where a correction routine is run when an error is detected. Let's code our first do while loop by changing our *while()* program. Follow these instructions:

1. Open the following IDE C++ workspace:
<https://ide.geeksforgeeks.org/0WjOL6TSh4>
2. Initialize counter to *inti = 11;*
3. Implement the *do{}* code block
4. Set *while(i < 11);*
5. Clean the data of any unneeded code and comments
6. Check the results



The screenshot displays a C++ IDE workspace. On the left, a code editor shows a C++ program with the following code:

```
1 #include <iostream>  
2 using namespace std;  
3  
4 int main() {  
5     //Initialize the counter, i  
6     int i = 11;  
7  
8     //Print explanation of program  
9     cout << "This program may or may not  
10    << "\nHello World!\n";  
11    //While i is less than 11, print "Hello"  
12    do {  
13        cout << "\n" << i << " Hello  
14        i++; //Unary counter  
15    }  
16    while (i < 11);  
17    return 0;  
18 }  
19  
20 }
```

On the right, the IDE interface shows the following information:

- Generated URL:** <https://ide.geeksforgeeks.org/t6coWCH7> (with a Copy button)
- Time(sec):** 0
- Memory(MB):** 3.2256820561055
- Output:** (with a Copy button)
This program may or may not print one
"Hello World"
11 Hello World

Do While Loop Demo exhibiting an exit controlled loop

This screenshot is of our do while loop demo. You should notice that,

i
despite *< 11* being *false* for the entire duration of the program,
the *do{}...* code block was executed. This is because the loop is an exit control loop: the condition is tested after the code block is executed. The code has the potential to execute. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/t6zcNK2Wj7>.

7.

See how the message was printed despite the conditions being false. You can apply it to data corrections in streams. Many error detection services pause the stream of data and a request for retransmission is sent. A *do{}...while* loop will allow the data stream to continue until an error is detected. These are the types of routines used by YouTube in server buffers. This shows how useful they are in controlling data. For single switching, we would use an *if()...else*.

If()...Else

So far we have used *if()...else* functions in our examples. They are easy to understand and their structure is pretty self-explanatory. An *if()...else* statement tests a condition and based on the result it executes one block of code or another. They are a combination of two separate distinct statements.

You can execute the *if()* function alone but the compiler expects an *else* statement. An *else* statement contains the code that accompanies the *if()* expression. An *else* statement can never be executed without the *if()* statement.

An *else if()* statement specifies another condition to test. It creates a chain of statements called nested *if-else* statements. If you are testing a lot of conditions, it is best to use a switch which we will discuss in the next section. Let's make our first nested *if-else* statement. Follow these instructions ("C++ | Nested Ternary Operator," 2018):

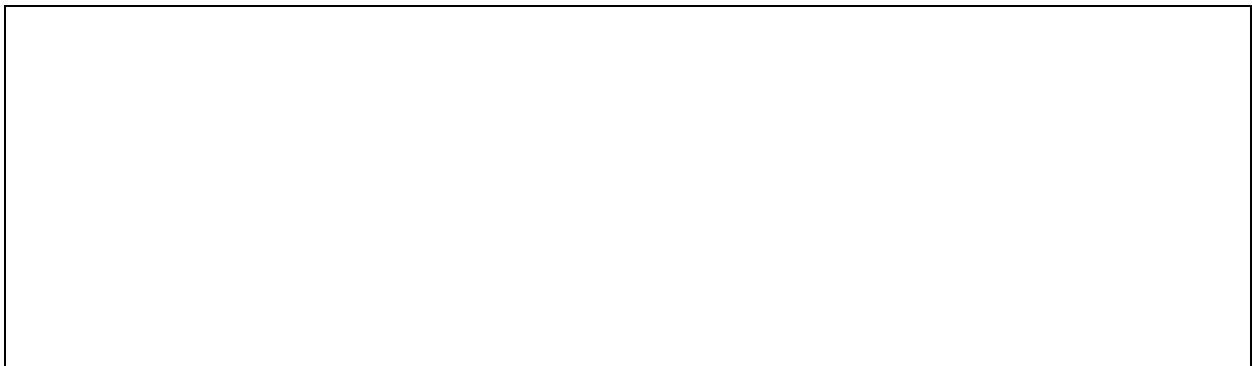
1. Open a new IDE C++ workspace
2. Print an explanation for the conditional operator

This program will go through a sequential list of numbers 2 through 4 and pick the greatest one using a nested *if()...else* statement

We expect a result of 4.

3. Implement a nested if-else

```
if(2 > 3) then print 2
elseif(3 > 4) then print 3
else print 4
```





Nested If-Else Demo

This screenshot is of our nested if-else demo. Just like our other demos, this program has no functionality. However, for our purposes, this is an excellent exercise for implementing and understanding

if()...else

statements. Despite us already knowing the answer, the *if()...else*

still

requires code blocks for each statement. The code blocks should also correspond to the condition in order to create a functional program that fulfills its stated purpose. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/89MlNvRWYU>.

Let's examine the exercise further. The nested *if-else* can be divided into two parts: the *if()* and *else – if()* statement that contains conditions and execution code. You notice that the blocks of code logically follow the state goal of the program, which is to find the largest number. It makes sense to print numbers in ascending order in case we change the conditional statement. Conditions within the *if()* statements and the blocks of code they contain are arranged in a way that allows the program to flow to the answer. For instance, if $(2 > 3)$ is hardcoded to the next nested statement. The structure followed the nested conditional operator deliberately.

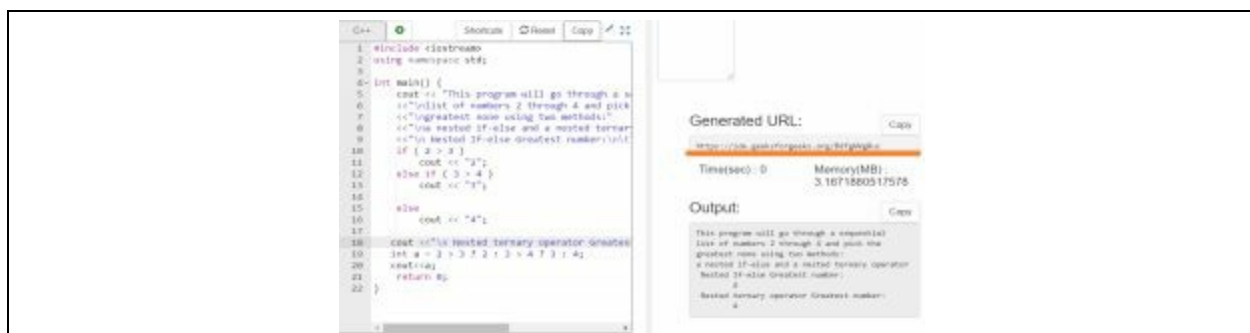
Nested Ternary If Else Operator

In the previous chapter, we talked about conditional operators, ternary operators that test conditions and execute an expression based on the results. We said the structure it uses is similar to an *if()...else* statement but more succinct. For instance, an *if()* statement tests a condition and executes one of

the two blocks of code depending on the result of the test. If ()...else statements can be nested, and conditional operators can also be nested. Let's compare the structures of both. Follow these steps to participate ("C++ | Nested Ternary Operator," 2018):

1. If you had closed the nested if-else demo, open up the saved IDE C++ workspace with our program at this URL: <https://ide.geeksforgeeks.org/89MINvRWYU>
2. Print an explanation for the program
 1. This program will go through a sequential list of numbers 2 through 4 and pick the greatest one using two methods: a conditional operator and a nested if...else statement
 2. We expect a result of 4.
3. Add a printed notification of the nested if-else
4. Print a notification for the conditional statement
5. Add the conditional operator code: declare the variable and execute the conditional operator

```
int a = 2 > 3 ? 2 : 3 > 4 ? 3 : 4;
```
6. Clean the code of any unnecessary lines and correct any annotations



Nested If-Else and Nested Ternary Operation Comparison

This screenshot is of our modified nested if-else demo. We modified our demo by implementing a nested ternary operator for side-by-side comparison. As you may have experienced, the ternary operator was much easier to implement, using fewer statements. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/BdfgWVg8uc>.

Let's talk about what we have just done there. In our previous discussion, we broke down the nested *if-else* statement into its components and a list of possible results. Even in the case where "2" and "3" were arbitrary values, if-else statements have to list all possible results to ensure great functionality regardless of the conditional statements results.

To see how conditional statement results of a nested *if-else match* with components of ternary operator, we have to take a close look at the ternary operator. Here is the syntax: $var = (expression)_1 ? [resultiftrue]_1 @ (expression)_2 ? [resultiftrue]_2 : [resultif false]_2$

Var is the variable used to print the result. The result of the nested if-else match expression before the ? indicators of the operator. The second expression executes when the first expression is false and corresponds to the *else if* () in the nested if-else. You can already see advantages to the compactness of ternary operators, but by their nature ternary operators are not suited for code blocks. With three or more expressions both get cluttered, so we need something else to handle multiple expressions: the switch statement.

Switch

As I have said, programmers need to strive for efficient, easily understandable code. It makes work easier for you and everyone else. Using switch over *if()..else* is one way you can achieve this, especially in circumstances where we are handling a lot of expressions. Ternary operators reduce lines of code, and it is because of this that they are not suited for situations where the code needs to span more than one line.

Switch statements offer a solution for both problems. Switch statements are multibranch statements that provide a clean way to execute a different block of code on one variable. Here is the syntax (Awasthi, n.d.):

```
switch(n)
{
    case1://codetobeexecutedifn = 1;
        break;
    case2://codetobeexecutedifn = 2;
        break;

                                default://codetobeexecutedifndoesn'tmatchanycases
}
```

Note: A switch can only evaluate an integer against case numbers. To add the conditional component, you must change the variable with an operation. The operator must produce an integer.

Let's create a switch demo practice the components of a switch:

1. Open a new IDE C++ workspace
2. Initialize a variable

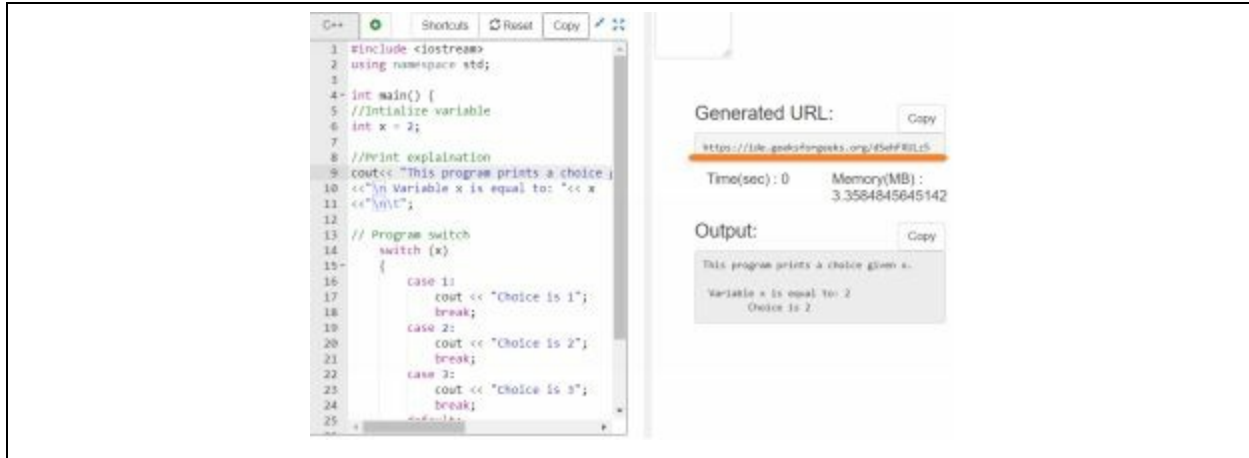
```
int x = 2;
```

3. Print an explanation of the program
4. Initialize the switch

```
switch(x)
```

5. Print out "Choice is 1" for case 1

6. Print out “Choice is 2” for case 2
7. Print out “Choice is 3” for case 3
8. Print out “Program Exit” for the default;



```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     //Initialize variable
6     int x = 2;
7
8     //Print explanation
9     cout << "This program prints a choice :
10    << "\n Variable x is equal to: "<< x
11    << "\n";
12
13    // Program switch
14    switch (x)
15    {
16        case 1:
17            cout << "Choice is 1";
18            break;
19        case 2:
20            cout << "Choice is 2";
21            break;
22        case 3:
23            cout << "Choice is 3";
24            break;
25    }
26    return 0;
27 }
```

Generated URL: <https://ide.geeksforgeeks.org/dSehFXULz5>

Time(sec): 0 Memory(MB): 3.3584845645142

Output:

This program prints a choice given x.

Variable x is equal to: 2
Choice is 2.

Switch Demo

This is a screenshot of our switch demo. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/dSehFXULz5>. We highly suggest using this demo to convert one of our previous exercises into a switch. One excellent candidate is our relational routine program. It had 6 if()... else statements! This comes out to 12 different scenarios. How could you convert this program into a switch? Hint: Consider creating a menu for each relational routine using the input function. You can study the relational routine for yourself at this URL: <https://ide.geeksforgeeks.org/dFD3cEk85y>.

Now that we have seen what decision making functions are and how they work, let's turn our attention to functions and their role in programming.

Chapter 6: Creating Functions

We talked about algorithms as a method of understanding problems and planning how your code will work to fix it. Algorithms receive inputs, perform actions, and then give a result which then causes a state change. A **function** is a block of code you can call multiple times without having to write the code again. The code is usually something that performs a desired task and has high re-usability. A function is composed of the header and a body. Here is how the syntax of a function looks:

```
[datatype]function_name(arguments)
    { //statements
        return0;
    }
```

Like you have seen, and like many other programming languages, C++ is mostly just a series of functions, statements, and operators with objects and other data-holding elements for in-between. Everything we have written so far has used the *main()* function which uses syntax similar to the one above. So functions aren't new to you.

Every function must return a data type, just like the *main()* function returns the *int* 0. If a function does not do this, (void) is used as the parameter return type. As you might have guessed, the *return* is the statement that terminates the function.

The inputs that the functions receive are called parameters or arguments. Not all functions will take inputs as we have seen with *main()*; regardless of this, you can still pass parameters into the function for it to run. This is how libraries pass objects into *main()*.

Create and Call a Function

To begin calling a function, you must declare it and define it before *main()*. This is because C++ executes sequentially. C++ practices take precedence and functions get priority. This is why you must declare them before *main()* so you can call them. They are also called this way because they make code more organized. Once a function is declared, we can define it anywhere outside of *main()*

Let's write a simple printing function so we can appreciate them more. Follow these instructions:

1. Open a new IDE C++ workspace

2. Declare your function

```
void myFunction();
```

3. Call your function in main ()

```
myFunction()
```

4. Define your function outside of main

```
{cout << "I just got executed!";}
```



Function Demo

This screenshot is of our function demo. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/b9iSDihApZ>. If you want more practice, we suggest using this exercise as a template to turn one of our past exercises into a function! Consider our exercise using *cin* and passing parameters to it. Any function with *cout* will do.

Separating the declaration and the definition falls under good programming practices. It makes it so you know all defined functions in the code that are available for calling. Having definitions at the bottom makes it easy to parse for errors within functions. Now let's turn our attention to parameter passing.

Parameter Passing

Functions are scripts that are given to the compiler. To complete the task and be useful, these functions need inputs to compute. To remind you, these inputs are called arguments or parameters. Programmers describe parameters differently based on where they originate or appear. Parameters that go into a defined function are called actual parameters. For instance, if you have a summing function called *mySum(x,y)*, when *main()* passes 2 and 3 into the functions those parameters are actual parameters. Conversely, x and y are formal parameters because they are variables that the data is going to be bound to. This applies to all other parameters before actual parameters are passed. There are two ways to pass parameters to functions (“Functions in C/C++,” 2015):

Pass by Value: Values of actual parameters are copied into and stored in the function’s formal parameters. Any changes made inside *main()* do not impact the actual parameters that were passed.

Pass by Reference: Both actual and formal parameters refer to the same locations. Therefore, any changes made inside the *main()* function will impact the actual parameters that were passed.

Let’s practice parameter passing by writing our own code. Follow these instructions:

1. Open a new IDE C++ workspace
2. Declare your function

```
int max( int x, int y)
```

3. Call your function in main ()
4. Initialize two actual variables

```
inta = 10,b = 20;
```

5. Initialize a variable and call the function

```
int m = max(a, b);
```

6. Print the result
7. End main()

8. Define your function outside of main
 - i. This function should use an `if()` statement to check if `x` is greater than `y`.
 - ii. If `x` is greater than `y`, then return `x`
 - iii. Otherwise, return `y`.



```
1 #include <iostream>
2 using namespace std;
3
4 //declare function
5 int max(int x, int y);
6
7 //Main
8 int main() {
9     int a = 20, b = 25;
10
11     // calling above function to find max of 'a'
12     int m = max(a, b);
13
14     cout << "m is " << m;
15     return 0;
16 }
17 //define function
18 int max(int x, int y)
19 {
20     if (x > y)
21         return x;
22     else
23         return y;
24 }
```

Generated URL: <https://ide.geeksforgeeks.org/tnsZczhX15> Copy

Time(sec): 0 Memory(MB): 3.2489796331787

Output: m is 25 Copy

Functions with Parameter Passing

This screenshot is of our function demo. You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/NnsZczhX15>. If you want more practice, we suggest using this exercise as a template to turn one of our past exercises into a function! Consider our exercise using `cin` and passing parameters to it. We'll discuss how to pass parameters in the next section, but trying it first on your own will help you grasp the concept.

In our example, we have used value passing well. The variables used for *main* (`a,b`) are in a different location than those used in *max()*, (`x, y`). You can see how this can be very useful in larger programs where several complicated computations might take place in *main()*. It is preferred to pass values because it ensures that actual variables remain present until the function is called again.

Using a reference to pass arguments has its own advantages, especially in larger programs. References to pass inessential values may speed up the program and make it responsive. Scripting for automating functions is more likely to use references to pass parameters since automated processes are

likely to happen repeatedly and quickly. These qualities are a selling point for reference-based structures. Let's now turn our attention to condensing and optimizing code.

Function Overloading

All defined objects and classes must have unique names. You cannot run a program that has two variables that have the same name or an error will be thrown. You might have seen this earlier while we're working on our routines and demos, especially when we had to modify a program. So keep this in mind: variables that are declared twice cause errors. But weirdly, multiple functions can have the same name as other functions as long as they have different parameters. This is only allowed in circumstances where the functions return different data types. It is better, given that, to overload a function to return multiple data types. This is what is called **function overloading**; a process where programmers combine functions to receive multiple data types. Why? Because it simplifies and consolidates the code. Take a look at the following code (“C++ Function Overloading,” n.d.):



The screenshot shows a C++ IDE with a code editor on the left and a console/output window on the right. The code in the editor defines two functions: `plusFuncInt` which takes two integers and returns an integer, and `plusFuncDouble` which takes two doubles and returns a double. The `main` function calls `plusFuncInt` with 0 and 5, and `plusFuncDouble` with 4.3 and 6.28, storing the results in `myNum1` and `myNum2` respectively. The console output shows the integer result 5 and the double result 10.58.

```
1 #include <iostream>
2 using namespace std;
3
4 int plusFuncInt(int x, int y) {
5     return x + y;
6 }
7
8 double plusFuncDouble(double x, double y) {
9     return x + y;
10 }
11
12 int main() {
13     int myNum1 = plusFuncInt(0, 5);
14     double myNum2 = plusFuncDouble(4.3, 6.28);
15     cout << "Int: " << myNum1 << "n";
16     cout << "Double: " << myNum2;
17     return 0;
18 }
```

Generated URL: <https://ide.geeksforgeeks.org/qkpHhZsqq5>

Time(sec): 0 Memory(MB): 3.3850370980835

Output:

```
Int: 5
Double: 10.58
```

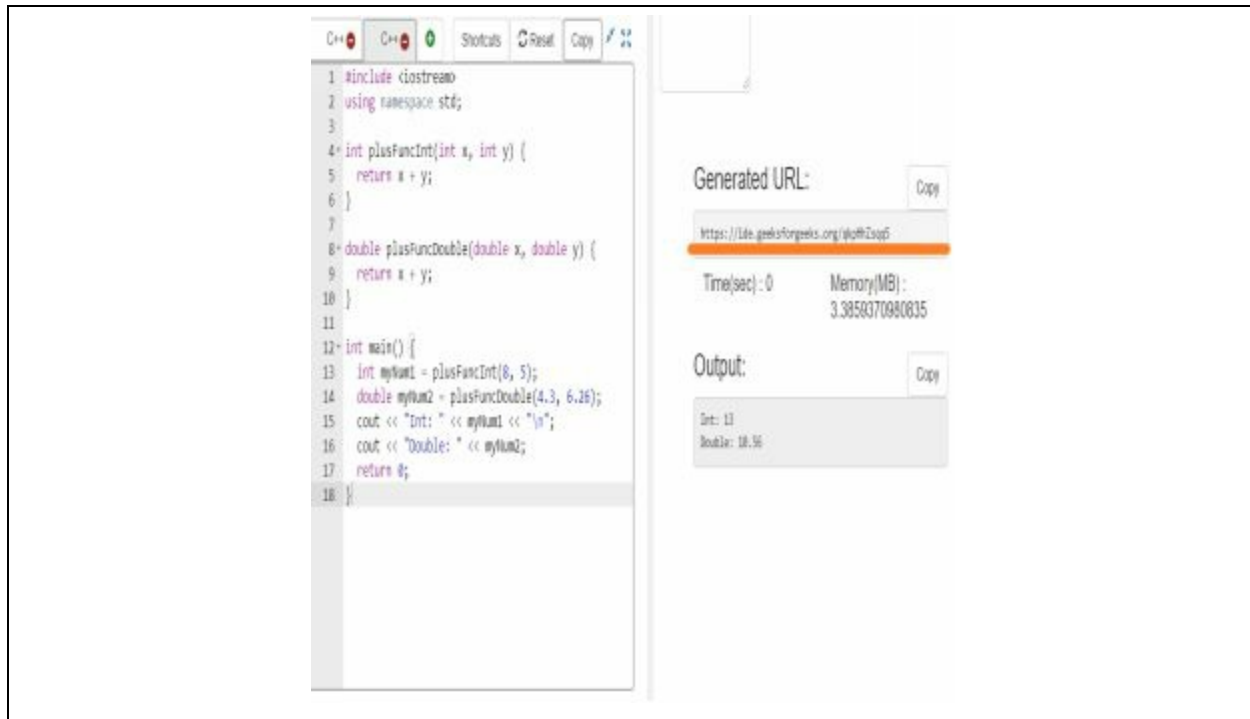
Program without function overloading

You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/qkpHhZsqq5>. The functions are declared and defined at the top of the file so that we can observe them better. You will be using this file to perform function overloading on your own. Remember that you can return to this workspace at any time if your code is damaged during editing.

The program above declares two separate functions that do similar computations. In a big program that is a waste of real estate and it is inefficient. It makes it harder to edit the code since either one of the functions are unlikely to be in the same place in a larger program. To make things more

efficient, we should consolidate these functions under one name, and “stack” the data types. Here’s how you can do this:

1. Open the saved IDE workspace at the following URL:
<https://ide.geeksforgeeks.org/qkpHhZsqq5>
2. Change double plusFuncdouble...to double plusFunc...
3. Call double plusFuncdouble... in place of double plusFunc...



Program function overloading

You can access a copy of our saved workspace using this URL: <https://ide.geeksforgeeks.org/e1qLzi3aGo>. We kept the functions at the top of the file for easier editing. This overloading exercise requires a few small minute detail changes in order for this file to work. Use our workspace to compare your file in the event you get any long-standing errors. Remember that you can return to the original workspace at any time if your code is damaged during editing. You can access the original workspace at this URL: <https://ide.geeksforgeeks.org/qkpHhZsqq5>.

When you consolidate similar functions, you clean up your code. It is important, especially when you have to debug the code, since the compiler will likely point to functions when errors occur. The compiler does this

because it is pointing to multiple libraries in the C++ code. Consolidating and overloading functions ensures that your compiler will have fewer functions to reference, making it easier for you to debug. Keep in mind that C++ is constantly updating so it becomes even easier to work with. New features are constantly added, and libraries are condensed and further organized. So an error in C++7 would look different than an error in C++14. Updates bring new, easy ways to perform mundane tasks. For instance, the copy constructor call combinations are considered more modern for indexing containers such as arrays. To stay updated and reap the benefits of new C++ developments, you have to keep practicing, testing yourself, and asking questions.

Conclusion and Final Notes

Learning programming is a cognitive journey, one that is stimulating, exciting, and sometimes frustrating. As we come to the end of this book, it is important to go over some of the important lessons we learned. They are also going to be accompanied by final thoughts and suggestions that are meant to equip you with all you need to continue growing in this field.

Expanding Your Practice: Preparing Your Coding Environment

We have used an online IDE to make it so that everyone has the same experience. If you want to save your work you can use the Run+URL button to save it in another URL. You can also bookmark all these for later reference using your saved IDE workspace and URLs listed in this book. Use them as a reference or to fix broken code. I have done my best to show you the best coding practices in all the exercises we did. You have seen that in C++ there are many ways of doing something, so use that. Don't be afraid to experiment and try new things, and read up on how to do other things you are interested in. The exercises in this book are not definitive answers and nothing like that exists in programming. So come to this book for reference and explore other ways of doing something. We have our codes in the index of this ebook for quick reference.

There are many programming environments and text editors that can support C++. Some will have multiple versions of C++. As of writing this, the most up to date version of C++ is 17 but the most supported is C++14. C++ 14 is the evolved version of C++ and it features more comprehensible language features. C++14 is more intelligible to humans than its predecessor and you can expect this trend to continue. The oldest, widely available C++ version in most programming environments is C++11.

I encourage you to set up a local IDE and work on it. It will allow you to grow faster and get accustomed to real-world scenarios. We have spoken of Code::Blocks for Microsoft and Xcode of Mac. If you have Linux there are standardized options you can use across all distros. Linux is a console-based environment that is highly preferred by software developers because console-based environments are entrenched with many built-in features that directly interact with programs.

There are many distros that provide a GUI-based desktop that is preferable for many users. Most of them are open source, free to download and install. We highly recommend CentOS, a free Linux distro server management. CentOS uses many of Red Hat Enterprise Linux (RHEL) components. It is the best distro for preparing to work in Red Hat server management

environments.

Coding Best Practices: Ownership of Learning and Collaboration

In our time together we have seen many techniques and best practices for software development. In the first chapter, we talked about how programs should fulfill their tasks with as little code as possible. Thankfully, with each new version of C++, more ways of doing this are added. The majority of beginner programming courses teach older, less efficient methods of implementing common programming tasks like parsing arrays, conditional statements, and calling functions. You will find most of these techniques are never used in actual programming situations. Most beginner courses are focused on equipping beginners with a strong grasp of the basics – the foundation of your knowledge is often as important as how you are going to progress. As you grow you will find more efficient ways of solving problems with your code, but this can only happen if you practice and study.

If you want to improve as a programmer you have to take ownership of your learning. Learn pre-development tools like algorithms. It is all about learning to fix problems. There are other pre-development tools you can try, like diagraming. Diagraming is good for individuals who are visual learners. All of these are great for growing as a programmer; it does not matter which you choose, as long as pre-developments tools are something you take advantage of.

Turning back to algorithms, more detailed algorithms will explain all sorts of data types required for inputs, outputs, and other data. The advantage of algorithms is that you can use them as a shorthand. You can also annotate and manage your code. These habits will help keep your code ordered and coherent to your team members. All these practices will help enhance how you see and think about programs.

Software developers work in teams, with several people working on the same piece of code at the same time. The importance of annotating your code becomes very apparent in such situations.

It also happens that working with others will accelerate your growth. When coding with others you learn from their experience and knowledge. As we have said, C++ has a variety of library objects, statements, and other tools. It

is as rich as vocabulary in human languages because there are many ways of doing the same things. Your way of fixing a problem will depend on your ability to see efficient ways of doing so. But ultimately all programming and software developments are creative endeavors. Working with others will increase your programming “vocabulary” – you will learn from how creatively they fix problems and you will find insights of your own in the process. You should always keep in mind that everyone can contribute something and what they contribute is heavily influenced by the amount of experience they have. So even if you are new to C++, you still have a unique experience and approach to solving problems that other people don’t have because you think differently. That in itself is valuable to any team, particularly if you learn to communicate your ideas well. I highly recommend you use the glossary as one of the ways to help you communicate clearly.

In addition to annotations, it is good practice to make your code as clean as possible. I have demonstrated how to comment lines of code to prevent breaking the code during editing. In your exercises, when you break the code beyond repair, you can always start over from our saved IDEs, or you can simply refresh the browser to start over (this will depend on the state you ran and generated the code). Only generate URLs when you need to save your code, not while you are developing.

We also stressed the importance of code structure. For instance, we talked about the importance of declaring and defining functions: the best practice is to declare your functions on the top and define them after the `main()` function. It is similar to declaring integers. The techniques ensure you have an inventory of functions, variables, and objects right at the beginning. It works like an index and can help you avoid definition errors.

Formatting and editing your code of unnecessary lines reduces the possibility of errors during development. You should always update your comments when you make a change. Unformatted code will not impact performance, the logic of the code will. It is a collaboration courtesy to format your code, not to mention how much easier it makes things for you. Other programmers on your team will be able to implement your code, or even improve it, and they will leave comments.

But these types of practices are not just good for you. As the world moves

forward it is becoming more and more complicated, so it is good for programs to explain themselves to the user. If your program prints, make sure it is easy to read. We have seen how this can be done in this book. Use escape sequences like `\n` to create new lines and `\t` to create horizontal table so there is enough whitespace to your code. But as a back-end programmer working with servers and other equipment, you may not have to print out as often as a front-end programmer who serves content to the client. The only users who may need to see your print outs are other technicians and administrators managing those machines through the terminal. Formatting practices are often underused in backend programming, but it is still a good practice to have. It is valuable to administrators and technicians, who do not have the resources to navigate esoteric compiler-generated errors that programmers are used to when debugging. So a program that is formatted and gives this information clearly in the terminal or consoles is important.

Take Away: Computer Science Concepts in C++

In addition to programming, we also explored some basic computer science concepts. We looked at the behavior of machines and their language. We talked about how learning programming is like learning a new language: both have syntax and rules that ensure communication is possible. C++ is a back-end server language that is used for managing servers that control oceans of data. Unlike high-level languages like Python, which are more human, C++ wants programmers to be more detailed in their code. Each variable must be unique in a function or we will have an error. All functions and statements are case sensitive and have specific rules that govern them. We have implemented many *if()...else* statements and made nested *if-else* statements. We saw that conditional statements like *if()* can be executed without the *else*, but a missing *else* statement can cause an error if the condition is false. And just as a missing verb indicates a sentence is incomplete, *else* statements cannot be implemented without an *if()*.

We saw how computers communicate in binary numbers using logical math. We saw bitwise function work at the bit level, completing logical math computations like AND, OR, and NOT. All the other C++ functions that are object-based must be translated by the compiler into machine language. But *bitwise* functions are faster because they skip that step and communicate directly to the machine. They are often deployed in competitive programming to make highly agile programs and routines. Outside of competitive programming, bitwise functions have some functionality in managing big data.

The challenge with *bitwise* functions is that they are very difficult for humans to understand because they are binary. To see what computers see we had a brief computer science lesson on binary and binary math. We made a “byte-sized” stream of register with our bitwise routine program and we made dynamic buffers to store bits for bitwise computations. Those virtual buffers and registers are how servers manage terabytes of data. Think about this; our buffers and registers were all one byte long, but a terabyte is 10^{12} times larger than our buffer: 1,000,000,000,000

bytes. These numbers are small for websites like
YouTube which bases its back- end computing on C
++. The data that YouTube deals with is astronomical.

In 2018 it was 5.25 zettabytes (10^{21} bytes). It was serving over 3.25 billion hours of content (“How much data does YouTube store? - Quora,” n.d.). Bitwise functions give these servers the ability to work with this load without interruption or massive delays. This is one of the best examples of the power of C++ and its ability to handle big complex systems. It is a useful language for programmers to learn. If you visualize these data streams as arrays of 8 slots, it will be easier for you to rationalize many bitwise operators, especially shift operators.

In addition to binary programming, we worked with different kinds of data and explored different aspects of object programming. We did plenty of exercises where we dealt with strings, for example. In fact, we used and deployed strings in every one of our exercises. We saw that strings are also arrays of data, and can be manipulated like objects except in a few ways. Strings have a length, size and can be parsed with various objects in the `<string>` library. Think of the copy constructor call combination we learned. To parse a container of any type, vector or array, you need a *for()* loop with an *Addressof* operator element. That task has a dedicated function in the `<string>` library where all you need is the index of the item you are parsing. If there is one thing you should take away from this book it is proficiency in handling strings: breakout escape sequences, handling strings through array indexing, and passing strings to different functions.

We also learned and discussed functionality and design considerations. We talked, for instance, about how and when to use loops. Our bank security example was best illustrated by our exit controlled loop example in a fail-close system. Banks, as we said, should always have their doors locked and closed; they fail-close. Most events in the bank require doors and exits to fail-close and the only time they should open it is in a fire. The system should keep everything in check, where a condition is tested AFTER the routine. The exit control loops are also useful in other areas, like in data control. YouTube does data corrections as a secondary routine of normal operations. This makes sure the system is always running, not hunkered by resend-data

requests.

As C++ is a backend-language it has to handle and work with multiple types of data. We have had a brief discussion about this when we spoke of vector-based images as an object class and their other object properties. Photo manipulation software like Photoshop is programmed in C++ where aspects like image manipulation take full use of the program's power. Vectors allow image manipulation without loss of quality, unlike .jpeg or .png that are arrays of pixels. Vectors like .svg or .pdf use vector containers to describe their pixel locations, which has more class features in C++ that allows preservation of the image.

This book is also meant to be reference, so I encourage you to use the glossary and the index to review topics. The glossary has all the terms bolded and some of the code snippets we looked at. It also includes the section where the terms are discussed in more detail.

Glossary

Term	Definition	Section
"+"	Binary, arithmetic operation that sums two variables. Also used to concatenate strings in the <string> library	Operation:String Concatenation
"++"	Increment operator used to increase the value of an integer by 1. This unary operator is often used to increase a variable within a loop.	Unary Operators
"--"	Unary operator that decreases a variable within a loop. This unary operator is often used to increase a variable within a loop.	Unary Operators
"-", Arithmetic Operator	Binary operator that subtracts one operand from another	Arithmetic Operators
"-", Unary Operator	Unary minus operator that is used to change the sign of a variable or argument. Performing a unary minus operation on a negative integer will make it positive and vice versa.	Unary Operators
"-=	The atomic subtraction used to combine the '-' and '=' operators. This operator first subtracts the current value of the variable on the left from the value on the right. It then assigns the result to the variable on the left. For example, x-=y would be a shorter way of writing x=x-y without having to use a separate arithmetic operator.	Assignment Operators
"!", Binary Operator	Binary logical "NOT" operator that returns a Boolean true if the conditions in consideration are not satisfied. If one of the conditions is true, then it returns a Boolean false.	Logical Operators
	NOT operator is used to reverse the Boolean logical state of an	

“!”,	operand. For example, if a variable x has a Boolean value of false, !x will be true.	Unary Operators
“!=”	“Not-Equal” operator that also checks whether two given operands are equal or not. However, this operator returns a Boolean true if the operands are not equal and returns false if the operands are equal.	Relational Operators
“*”	Binary operator that multiplies two operands. The order of the operands does not matter.	Arithmetic Operators
“*=”	Operator for the atomic multiplication, used to combine the ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on the left to the value on the right and then assigns the result to the variable on the left. For example, x*=y would be a shorter way of writing x=x*y without having to use a separate arithmetic operator.	Assignment Operators
“/”	Binary division “forward slash” operator that divides the first operand into the second.	Arithmetic Operators
“/=”	Atomic division operator is a combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on the left by the value on right and then assigns the result to the variable on the left. For example, x/=y would be a shorter way of writing x=x/y without having to use a separate arithmetic operator.	Assignment Operators
“?” Escape Sequence	Used for question marks.	Using Escape Sequences
“\” Escape Sequence	Used for single quotes.	Using Escape Sequences
“\”” Escape Sequence	Used for double quotes.	Using Escape Sequences
“\\” Escape Sequence	Used for backslashes.	Using Escape Sequences

“\f” Escape Sequence	Stands for “form feed” and is used to go to the next “page”.	Using Escape Sequences
“\n” Escape Sequence	Stands for “line feed” and is used to go to the next line.	Using Escape Sequences
“\t” Escape Sequence	Stands for “horizontal tab” and adds 5 spaces horizontally.	Using Escape Sequences
“\v” Escape Sequence	Stands for “vertical tab” and is used for spacing in vertical languages.	Using Escape Sequences
“&” Unary Operator	Addressof operator that is used to point to the address of a variable.	Unary Operators
“&”, Binary Operator	Bitwise AND that takes two numbers as operands and does AND on every bit within the stream of the two numbers. The result stream of AND is 1 only if both bits are 1.	Bitwise Operators
“&&”	Binary logical “AND” operator that returns a Boolean true when both the conditions in consideration are satisfied. Otherwise, it returns false.	Logical Operators
“%”	Binary modulus operator that returns the remainder from dividing the first operand into the second.	Arithmetic Operators
“^”	Bitwise XOR that takes two numbers as operands and does XOR on every bit within the stream of the two numbers. The result stream of XOR is 1 if the two bits are different.	Bitwise Operators
“+=”	Operator for atomic addition, used to combine the ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on the left to the value on the right. It then stores the result to the variable on the left. For example, x+=y would be a shorter way of writing x=x+y without having to use a separate arithmetic operator.	Assignment Operators

“<”	“Less-than” operator that checks whether the first operand is less than the second operand. If so, this operator returns a boolean true. Otherwise, it returns false.	Relational Operators
“<<”	Left shift operator that takes two numbers and left shifts the bits of the first operand. The second operand is used to determine the number of places to shift.	Bitwise Operators
“<=”	“Lesser-than or equal-to” operator that checks whether the first operand is less than or equal to the second operand. If so, this operand returns a Boolean true. Otherwise it returns false.	Relational Operators
“=”	Equal assignment operator, used to assign the value on the right to the variable on the left.	Assignment Operators
“==”	“Equal” operator that checks whether two given operands are equal. If the operands are equal, it returns a Boolean “true”, if not it returns false.	Relational Operators
“>”	“Greater-than” operator that checks whether the first operand is greater than the second operand. If so, this operator returns a Boolean true. Otherwise, it returns false.	Relational Operators
“>=”	“Greater-than or equal-to” operator that checks whether the first operand is greater than or equal to the second operand. If so, this operand returns a Boolean true. Otherwise, it returns a Boolean false.	Relational Operators
“>>”	Right shift operator that takes two numbers and right shifts the bits of the first operand. The second operand is used to determine the number of places to shift.	Bitwise Operators
“ ”	Bitwise OR that takes two numbers as operands and does OR on every bit within the stream of two numbers. The result stream of OR	Bitwise Operators

	is 1 if any of the two bits is 1.	
“ ”	Binary logical “OR” operator that returns a Boolean true when one (or both) of the conditions in consideration are satisfied. If none of the conditions are satisfied, it returns false.	Logical Operators
“~”	Bitwise NOT operator that takes one number and switches the state of all the bits (1s to 0s, 0s to 1s).	Bitwise Operators
“bool” Keyword	Datatype keyword that is short for “Boolean” and stores values with two states: true or false. Has a size of 1 byte. these values can be expressed as either “0/1” or “false/true” with a manipulator.	Basic Data Types
“boolalpha/noboolalpha” Manipulator	Input-Output manipulator that switches between using “0/1” to “false/true” for Boolean values.	Using Endl: Input/Output Manipulators
“char” Keyword	Data type keyword that is short for “character” and stores single characters regardless of their capitalization. Has a size of 1 byte. Char values are surrounded by single quotes.	Basic Data Types
“cout” object	C++ programming object part of the <iostream> library that allows a program to print out values and text using the "<<" operator. This object must be defined with a datatype, usually "std".	Using Cout
“double” Keyword	Data type keyword that is used to store numbers with decimals with 15 decimal digits. Has a size of 8 bytes. This data type can hold more decimal numbers and is preferred for mathematical calculations.	Basic Data Types
“else”	Statement is used to specify a code in the event that its accompanying if() expression is false. Else cannot be used without if ().	Loops
“endl” Manipulator	Input-Output manipulator that outputs “\n” and flushes the output	Using Endl: Input/Output Manipulators

	stream.	
“ends” Manipulator	Input-Output manipulator that outputs “\0” [zero].	Using Endl: Input/Output Manipulators
“float” Keyword	Keyword data type that stores numbers with decimals with 7 decimal digits. Has a size of 4 bytes. This data type is preferred for holding monetary numbers.	Basic Data Types
“flush” Manipulator	Input-Output manipulator that flushes the output stream.	Using Endl: Input/Output Manipulators
“get_money” Manipulator	Input-Output manipulator that receives an input as a monetary value. This manipulator works in C++11 only and may cause errors in outdated C++ programming environments.	Using Endl: Input/Output Manipulators
“get_time” Manipulator	Input-Output manipulator that receives an input as a date/time value according to a specified format. This manipulator works in C++11 only and may cause errors in outdated C++ programming environments.	Using Endl: Input/Output Manipulators
“int” Keyword	Data type keyword that is short for “integer” and stores whole numbers. Has a size of 4 bytes. This keyword includes positive and negative integers.	Basic Data Types
“put_money” Manipulator	Input-Output manipulator that formats and outputs a monetary value. This manipulator works in C++11 only and may cause errors in outdated C++ programming environments.	Using Endl: Input/Output Manipulators
“put_time” Manipulator	Input-Output manipulator that receives an input as a date/time value according to a specified format. This manipulator works in C++11 only and may cause errors in outdated C++ programming environments.	Using Endl: Input/Output Manipulators
“quoted” Manipulator	Input-Output manipulator allows you to insert and extract quoted strings with embedded spaces. This	Using Endl: Input/Output Manipulators

	manipulator works in C++14 only.	
“showbase/noshowbase” Manipulator	Input-Output manipulator specifically for mathematical outputs that controls whether a prefix is used to indicate a numeric base.	Using Endl: Input/Output Manipulators
“showpos/noshowpos” Manipulator	Input-Output manipulator that controls whether the “+” sign is used to indicate non-negative numbers.	Using Endl: Input/Output Manipulators
“uppercase/nouppercase” Manipulator	Input-Output manipulator that controls whether uppercase characters are used with some output formats.	Using Endl: Input/Output Manipulators
“using namespace std;” Statement	Line of code frequently used to unilaterally declare the "std" datatype for iostream library objects and manipulators. This unilateral declaration makes the code easier to see. However, this is discouraged outside of practice because it can make string handling ill-defined in programs, thus, causing errors that are difficult to mitigate. Exercises in this book will avoid using this statement.	Omitting Namespace
[string_name].length() Attribute	Coding syntax for calling the "length" string object attribute .	String Objects: Length() Attribute
Actual Parameters	Parameters coming into a defined function.	Parameter Passing
Algorithm	An organizational tool used to plot out the aspects of a program including (1) data in, if applicable, (2) operations performed on declared variables, and (3) the result of running the program.	Principles of Programming
Arithmetic operators	Binary and unary operations that are used to perform common mathematical operations.	Arithmetic Operators
Assembly language	Describes a low-level programming language that requires a compiler to convert it into machine code.	Chapter 1: Setting up a C++ Development Environment

Assignment Operators	Used to assign a value to a variable and change values on the fly, reducing the number of instructions in a program.	Assignment Operators
Atomic Operations	Operations that combine assignments with other operations. Atomic operations allow programmers to manipulate a value stored within a variable and reassign it immediately.	Assignment Operators
Base-10	Number system used by human beings assign place value to numerals. Base-10 is also known as the decimal system because a digit's value in a number is determined by where it lies in relation to the decimal point. The value is multiplied by a base power of 10, where each point away from the left of the decimal is $10^{(n+1)}$ and each point to the right of the decimal is $10^{(n-1)}$.	Bitwise Operators
Binary Number System	See Base-2	Bitwise Operators
Binary Operator	Classification of operators that execute with two operands.	Chapter 4: Operations in C++
Bitwise Operators	Logic operators that operate on numbers at the binary level.	Bitwise Operators
Bluefish	A free software advanced text editor with a variety of tools for programming in general and the development of dynamic websites.	Setting up the Text Editor
Byte	Measurement of data composed of 8 bits. A byte can represent up to 225 numbers (0b 1111111).	Unary Operators
C++14	Advanced version of C++ that makes this version slightly more intelligible for humans to program in. "Standard C++" usually refers to C++11, also known as C++0x.	Chapter 1: Setting up a C++ Development Environment.
CentOS	A free, community-supported. Linux distribution operating system that is functionally compatible with Red Hat Enterprise Linux (RHEL).	Linux Compiler Installation

Code	See Executable file	Chapter 1: Setting up a C++ Development Environment
Code::Blocks	A free open source IDE designed for C++. This IDE supports many C++ compilers including GCC, Clang, and Visual C++ in Microsoft. This IDE can program in C++, C, and Fortran.	Windows IDE Installation
Command-Line Interface (CLI)	Method of interacting with a computer program where the user controls the program with entering lines of code.	Setting up a Text Editor
Comments	Lines of code that are ignored by the compiler and indicated by a double slash at the beginning of the line in C++. Comments are used to annotate code and leave notes to guide other programmers studying your code.	Overview of C++ Syntax
Compiled Language	See Low-Level Language	Chapter 3
Concatenation, String	A common programming feature that allows programmers to dynamically control text by fielding content from a source and serving it within their program. In C++ String Concatenation is done through "+", an arithmetic operator.	Operation:String Concatenation
Conditional Operator	Boolean logic based operator with three operand expressions. The outcome of the conditional operator depends on the first expression.	Ternary Operators
Conditional Short-Circuit	When program developers use logical operators and conditional statements to control how a program behaves.	Logical Operators
Copy Constructor Call	Method of searching and printing indexes. Consists of for() loop with an index variable with an	Unary Operators

	Addressof operator, and a container object with an index such as an array.	
Counter	An arbitrary variable that is used in conjunction with an arithmetic unary operator to incrementally loop an indicated expression.	Loops
Data	Operands, or the various kinds of data that the functions and statements are acting on	Overview of Syntax
Data Type	Attribute of a variable which tells the compiler or interpreter how the programmer intends to use the data. Common data types include real numbers, integers, and "true-false" Boolean variables.	Chapter 3
Debugging	Process of location and removing computer program errors and abnormalities, also known as "bugs." As a general rule, smaller, modularized code that calls on functions is easier to debug than larger files.	Overview of Syntax
Decimal Number System	See Base-10	Bitwise Operators
do {...while()} Loop	Exit controlled loop that tests a single condition and loops a block of code until the condition is met.	Loops
else if()	Conditional statement combination used to nest conditional if()... else statements. Programmers are advised to use a switch if there are numerous conditions to test.	Loops
Escape Sequence	Special in-string character combinations that are used to represent certain special characters within strings and character streams. Escape sequences allow programmers to include symbols in strings without confusing the compiler.	Using Escape Sequences
Executable File	List of instructions that are used to perform the operations of a program.	Chapter 1

File Extension	File suffix that informs the code compiler what language the executable file is in. The C++ language is indicated by ".cpp" or ".hpp" extensions.	Chapter 1
First Principal of Programming	A program must be designed to complete a task in the smallest number of functions possible. This principle reduces overhead and increases performance of a program.	Principles of Programming
for() Loop	Entry controlled loop that has three inputs: an initialization expression, a test expression and an update expression. The for loop will execute its block of code until the test expression reports false.	Loops
Formal Parameters	Variables used to define the initial arguments, and any arguments in the function before the actual parameters are passed.	Parameter Passing
Function	A block of code that takes inputs to perform an action and runs when called.	Chapter 6: Creating...
Function Overloading	Programming technique of consolidating code through stacking several data types onto one function.	Function Overloading
Functionality	How program features and utilities can be implemented in a code to achieve a specific result.	Chapter 4: Operations in C++
Functions	Code that encapsulates instructions that may take inputs and output some result. Every function "[function-name]()", where [function_name] is some arbitrary function, is always followed by curly brackets {} and every explicitly stated instruction within the curly brackets will be executed by the compiler.	Overview of Syntax
Gedit	Free, open-source, general-purpose text editor that includes tools for editing source code and structured text such as markup languages.	Linux Compiler Installation

getline() function	C++ function from the <string> library that extracts all characters from an input string into a string array.	Cin and getline() function
GNU Collection Compiler (GCC)	An open source compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and is the standard compiler for most Linux projects.	Linux Compiler Installation
Graphical user Interface (GUI)	Method of interacting with a computer program that allows users to interact with graphical icons and visual indicators such as secondary notation instead of inputting commands.	Setting up Text
Header File Library	Lines that start with the pound sign (#) that are used by the compiler to call library functions.	Overview of Syntax
if() else Statement	Conditional statement function combination that tests a single condition and executes explicitly stated blocks of code based on the results of the test.	Loops
if()... else statement	A nested function that executes all instructions within its curly brackets “{}” given a specified condition is met. Otherwise, it executes the items with the enclosed else {} statement. These statements always deploy a relational operand to test the condition.	Relational Operators
Integrated Development Environment (IDE)	Programming environment used to write code, test for errors and translate a program.	Chapter 1
Linux distribution (distro)	An operating system made from a Linux kernel based software collection and package management system for installing additional software.	Linux Compiler Installation
	Physical electronic circuits that	

Logic Gates	have one or more inputs while only having one output. AND gates only pass a “1” when all inputs are “1”; OR gates pass a “1” as their output when they receive a “1” at any of their inputs; NOT gates pass a “1” when they receive a “0” at their input.	Logical Operators
Logical Operators	Binary operators used in combination with the relational operators combine two or more conditions to describe a specific constraint.	Logical Operators
Loops	Function that execute a block of code as long as a specified condition is reached	Loops
Low-Level Language	A programming language that closely follows a computer's instruction set architecture—commands or functions in the language map closely to processor instructions. This reduces the overhead for running the program as the compiler takes less resources to translate low-level languages into machine code.	Chapter 2
Machine Language	Coding language that is directly compiled and executed by a CPU. See "Assembler Language"	Chapter 1
Manipulator, Input/Output	Helper functions within the C++ iostream library that make it possible to control input/output streams using the "<<" operator or the ">>" operator.	Using endl: Input/Output Manipulators
Notepad ++	Text editor and source code editor for use with Microsoft Windows.	Setting up Text
Object	A general computing term that describes a method that the computer uses to manage data.	First Program: Output and Basic Strings
Operand	Operation component that describes the value, object, or variable being operated on.	Note: Operand...
Operations	Features such as control loops that implement decision-making in	Chapter 4: Operations in C++

	programs.	
Operator	<p>Operation component that describes the symbol indicating the sort of operation being carried out.</p> <p>Operators are described by how many operands they can operate on at a time: unary, binary, and ternary.</p>	Note: Operand...
Organization Identifier	<p>Proprietary Apple naming convention that is used to create a unique ID for programs across various Apple databases: the Apple Developer Website, and iCloud Container, iTunes connect portal in the Appstore. The organization identifier is used as the first argument of a “reversed domain” that ensures that all developed projects can be streamlined into the proprietary macOS framework.</p>	Mac IDE Installation
Parameter Pass by Reference	<p>Both actual and formal parameters refer to the same locations.</p> <p>Therefore, any changes made inside the main() function will impact the actual parameters that were passed.</p>	Parameter Passing
Parameter Pass by Value	<p>Values of actual parameters are copied into and stored in the function’s formal parameters. Any changes made inside main() does not impact the actual parameters that were passed.</p>	Parameter Passing
Red Hat Enterprise Linux (RHEL)	<p>A commercial Linux distribution developed by Red Hat for the commercial market.</p>	Linux Compiler Installation> Note: there are many distributions...
Register	<p>An array of binary storage where each bit gets its own place in the array. Registers are used to calculate bitwise operators.</p>	Bitwise Operators
Relational Operators	<p>Binary operators that are used to compare two values and return a boolean response.</p>	Relational Operators
Script	<p>Supplementary list of commands executed by certain programs or scripting engines.</p>	Chapter 6: Creating...

See Base-2	Number system used by machines to calculate computations. Also known as the binary number system. binary numbers are represented as a stream of bits, where each singular bit can be one of the two states: 1 or 0. The position of the 1 bit in the stream determines the value of the number.	Bitwise Operators
Server-side programming	Writing code that negotiates and delivers content to a dynamic, or changing website. C++ is, majorly, a server side language.	Mac IDE Installation
Service Level Agreement (SLA)	Term in IT management that describes a commitment between the technology administrator, the one providing the service, and a client.	Overview of Syntax
sizeof()	Unary operation that queries the size of an object and delivers its size in bytes.	Unary Operators
Software Development Kit (SDK)	A set of tools used for developing applications provided by hardware and software providers.	Mac IDE Installation
Statements	General term to describe the beginning line of code with specific instructions that the compiler recognizes outside of functions. How these are highlighted in language syntax environments depend on the type of the statement and the conventions of the programming environment.	Overview of Syntax
String Array	An orderly arrangement of characters where each character of the string is indexed in its own numerical location.	String Indexes and Arrays
Strings	A computer programming data type that represents a sequence of characters, either as a literal constant or as some kind of variable.	Advanced Strings

switch () Statement	Multibranch statement used to execute different code blocks based on the value of an evaluated expression	Loops
Ternary Operator	Classification of operators that accept more than 2 operands, or conditional operators with several arguments.	Ternary Operator
Text Editor	Program that enables editing of the code.	Chapter 1
Unary Operator	Classification of operators that execute with a single operand.	Unary Operator
Variables	A concept borrowed from mathematics to describe a symbolic value who's associated value can be changed and operated upon. Variables are used to control a program through operational functions.	Chapter 3
while() Loop	Entry controlled loop that tests a single condition and loops a block of code until the condition is met.	Loops
Xcode	A free IDE software development suite for the macOS. This IDE has utilities for developing C++ based programs for MacOS oriented operating systems like tvOS for Apple TV, watchOS for Apple watches, and iPadOS for iPad tablets.	Mac IDE Installation

Index

For your convenience, all coding program exercises have been posted here.

“Hello World” program in GeeksforGeeks IDE

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
using namespace std;
// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    cout<<"Hello World";
    return 0;
}
```


Using in-string escape sequences to delimit and manipulate the output

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
using namespace std;
// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    cout<<"Hello World!\n";
    cout<<"I am learning C++";
    return 0;
}
```

Using a input/output manipulator to delimit and manipulate the output

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
using namespace std;
// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    cout<<"Hello World!\n";
    cout<<"I am learning C++";
    return 0;
}
```

Testing Stability: Using a comment to remove “using namespace”

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
//using namespace std;
// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    cout<<"Hello World!" << endl;
    cout<<"I am learning C++";
    return 0;
}
```

Declaring the iostream objects and manipulators

With namespace

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    std::cout<<"Hello World!"
    << std::endl;
    std::cout<<"I am learning C++";
    return 0;
}
```

Without namespace

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
//using namespace std;
// main function -
// where the execution of program begins
int main()
{
```

```
// prints hello world
cout<<"Hello World!" << endl;
cout<<"I am learning C++";
return 0;
}
```

Declaring int numbers

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>
// main function -
// where the execution of program begins
int main()
{
    //Declares an int 'myNum'
    int myNum = 10;
    // prints hello world + myNum
    std::cout<<"Hello World!\n\t";
    std::cout<< myNum;
    std::cout<<" is my number.";
    return 0;
}
```

Summing inputs and printing the result

Inputs

11

12

Code

```
#include<iostream>
```

```
int main()
```

```
{
```

```
    // Explains program to user
```

```
    std::cout<<"Welcome! Enter two numbers and \nI will sum them.\n";
```

```
    //Declares input variables
```

```
    int x, y;
```

```
    //Declares output variable
```

```
    int sum;
```

```
    //Prompts user for inputs
```

```
    std::cout<<"Type a number: ";
```

```
    std::cin >> x;
```

```
    std::cout << x <<"\n";
```

```
    std::cout << "Type another number: ";
```

```
    std::cin>> y;
```

```
    std::cout << y <<"\n";
```

```
    //Operates on the variables
```

```
    sum= x+y;
```

```
    //Prints results
```

```
    std::cout<<"Sum is: "<<sum;
```

```
    return 0;  
}
```


Using Int with Float or Double

Inputs

22

7

Code

```
#include<iostream>
```

```
int main()
```

```
{
```

```
    // Explains program to user
```

```
    std::cout<<"Welcome! Enter two numbers and \nI will divide them.\n";
```

```
    //Declares input variables
```

```
    int x, y;
```

```
    //Declares output variable
```

```
    float result;
```

```
    //Prompts user for inputs
```

```
    std::cout<<"Type a number: ";
```

```
    std::cin >> x;
```

```
    std::cout << x <<"\n";
```

```
    std::cout << "Type another number: ";
```

```
    std::cin>> y;
```

```
    std::cout << y <<"\n";
```

```
    //Operates on the variables
```

```
    result= x/y;
```

```
    //Prints results
```

```
    std::cout<<"Result is: "<<result;
```

```
    return 0;  
}
```

Comparing Float and Double

```
#include <iostream>
int main()
{
    //Declares operation variables
    float x=27, y=7;
    double xd=22, yd=7;
    //Declares output variable
    float result;
    double resultd;
    bool test;

    //Operates on the variables
    result= x/y;
    resultd= xd/yd;
    test=(result==resultd);

    //Prints results
    std::cout<< "True or False?: \n\tfloat(" << x << "/" << y <<")";
    std::cout<< " is equal to double(" << xd << "/" << yd <<")\n";
    //std::cout<<"Is Float is Equal to Double?: ";
    std::cout << std::boolalpha
    << "\t"<<test;
    return 0;
}
```

String Concatenation

```
#include<iostream>
//Call the library for string handling
#include <string>
int main()
{
    //Declare variables and complete operation
    std::string firstName = "John ";
    std::string lastName = "Doe";
    std::string fullName = firstName + lastName;
    // prints results
    std::cout<<"Hello "
    << fullName <<". ";
    return 0;
}
```

Print fullName.Length()

```
#include<iostream>
//Call the library for string handling
#include <string>
int main()
{
    //Declare variables and complete operation
    std::string firstName = "John";
    std::string lastName = "Doe";
    std::string fullName = firstName + " "+lastName;
    // prints results
    std::cout<<"Hello "
    << fullName <<".\n";
    std::cout<<"Your full name is "
    <<fullName.length()
    <<" characters long.";
    return 0;
}
```

Access fullName String Array

```
#include<iostream>
//Call the library for string handling
#include <string>
int main()
{
    //Declare variables and complete operation
    std::string firstName = "John";
    std::string lastName = "Doe";
    std::string fullName = firstName + " "+lastName;
    // prints results
    std::cout<<"Hello "
    << fullName <<".\n";
    std::cout<<"The third character of your full name \nis \""
    <<fullName[2]
    <<"\'. ";
    return 0;
}
```

String Concatenation with cin

Input

John

Doe

Code

```
#include<iostream>

//Call the library for string handling
#include <string>

int main()
{
    //Declare variables
    std::string firstName;
    std::string lastName;

    //Prompt user for name
    std::cout<<"What is your name?\n";
    std::cin>> firstName;
    std::cout<< firstName <<"\n";
    std::cin>> lastName;
    std::cout<< lastName <<"\n";
    //Calculate operation
    std::string fullName = firstName + " " + lastName;
    // prints results
    std::cout<<"\nHello "
    << fullName << ".";
    return 0;
```

}

String Termination with cin

Input

John Doe

Code

```
#include<iostream>

//Call the library for string handling
#include <string>

int main()
{
    //Declare variables
    std::string fullName;

    //Prompt user for name
    std::cout<<"What is your full name?\n";
    std::cin>> fullName;
    // Greet user
    std::cout<<"\nHello "
    << fullName << ".";
    return 0;
}
```

Arithmetic Routine Program

```
#include<iostream>

int main()
{
    //Declare variables
    int a=25, b=3;
    int result;

    // printing a and b
    std::cout<< "This program will go through an\arithmetic routine with the
following\nvariables where: "
    <<"a is "<<a<<" and b is "<<b<<"\n";

    // addition
    result = a + b;
    std::cout << "\ta+b is: "<< result << "\n";

    // subtraction
    result = a - b;
    std::cout << "\ta-b is: "<< result << "\n";

    // multiplication
    result = a * b;
    std::cout << "\ta*b is: "<< result << "\n";

    // division
```

```
result = a / b;
std::cout << "\ta/b is: " << result << "\n";

// modulus
result = a % b;
std::cout << "\ta%b is: " << result << "\n";
        return 0;
}
```

Relational Routine Program

```
#include<iostream>
int main()
{
    //Declare variables
    int a=25, b=3;

    // print an explanation of the program
    std::cout<< "This program will go through an\nrelational routine with the
following\nvariables where: "
    // print a and b
    <<"a is "<<a<<" and b is "<<b<<"\n";

    // greater than example
    if (a > b)
        std::cout << "\ta is greater than b\n";
    else
        std::cout << "\ta is less than or equal to b\n";

    // greater than equal to
    if (a >= b)
        std::cout << "\ta is greater than or equal to b\n";
    else
        std::cout << "\ta is lesser than b\n";

    // less than example
    if (a < b)
        std::cout << "\ta is less than b\n";
    else
        std::cout << "\ta is greater than or equal to b\n";

    // lesser than equal to
    if (a <= b)
        std::cout << "\ta is lesser than or equal to b\n";
```

```
else
    std::cout << "\ta is greater than b\n";

// equal to
if (a == b)
    std::cout << "\ta is equal to b\n";
else
    std::cout << "\ta and b are not equal\n";

// not equal to
if (a != b)
    std::cout << "\ta is not equal to b\n";
else
    std::cout << "\ta is equal b\n";
    return 0;
}
```

Logical Routine Program

```
#include<iostream>

int main()
{
    //Declare variables
    int a=5, b=0, c=12, d=24;

    // print an explanation of the program
    std::cout<< "This program will go through a\nlogical routine with the
following\nvariables where: "

    // print a and b
    <<"a is "<<a<<" and b is "<<b<<"\n"
    <<"c is "<<c<<" and d is "<<d<<"\n";

    // logical AND example
    if (a>b && c==d)
        std::cout << "\ta is greater than b AND c is equal to d\n";
    else
        std::cout << "\t AND condition not satisfied--\n"
        << "\t\t(a>b && c==d)\n";

    // logical OR example
    if (a>b || c==d)
        std::cout << "\ta is greater than b OR c is equal to d\n";
    else
        std::cout << "\tNeither a is greater than b nor c is equal "
```

```
<< " to d\n";
```

```
// logical NOT example
```

```
if (!b)
```

```
    std::cout << "\tb is zero\n";
```

```
else
```

```
    std::cout << "\tb is not zero\n";
```

```
        return 0;
```

```
}
```

Short Circuit Detection Program

```
#include <iostream>
using namespace std;
int main() {
    //Declare two variables
        int a=10, b=4;
        //Print explanation of the program
        cout<< "This This program will test if the OR gate \n"
        <<"or the AND gate shorts when a is equal to b. \n";
        cout<< "\tCurrent value of a: "
        <<a <<"\n";
        cout<< "\tCurrent value of b: "
        <<b <<"\n";
    bool resAND = ((a == b) && cout << "--- OR gate short circuited.\n");
    bool resOR = ((a == b) || cout << "--- AND gate short circuited.\n");
        return 0;
}
```


Bitwise Routine with Register Printing

```
#include <iostream>
#include <bitset>
using namespace std;
int main() {
    //Declare operand numbers
    unsigned int a = 60; // 60 = 0011 1100
    unsigned int b = 13; // 13 = 0000 1101
    //Declare register c where calculations will take place
    int c = 0;

    //Print operand registers
    cout<<"Operand registers a: "<< a
    <<" and b: " << b
    << "\n\tregister a: -0b "<< bitset<8>(a)
    << "\n\tregister b: -0b "<< bitset<8>(b)<< endl ;

    //Bitwise AND
    c = a & b;          // 12 = 0000 1100
    cout << "Line 1 - Value of c is : " << c
    //Print register
    << "\n\t AND registers a&b: -0b "<< bitset<8>(c)<< endl ;

    //Bitwise OR
    c = a | b;          // 61 = 0011 1101
    cout << "Line 2 - Value of c is: " << c
```

```

//Print register
<< "\n\t OR register a|b:  -0b "<< bitset<8>(c)<< endl ;
//Bitwise XOR
c = a ^ b;          // 49 = 0011 0001
cout << "Line 3 - Value of c is: " << c
//Print register
<< "\n\t XOR register a^b:  -0b "<< bitset<8>(c)<< endl ;
//Bitwise NOT
c = ~a;             // -61 = 1100 0011
cout << "Line 4 - Value of c is: " << c
//Print register
<< "\n\t NOT register ~a:    -0b "<< bitset<8>(c)
<< "\n\t Compare to register a: -0b "<< bitset<8>(a)<< endl ;

//Left Shift
c = a << 2;         // 240 = 1111 0000
cout << "Line 5 - Value of c is: " << c
//Print register
<< "\n\t Left Shift register a<<2:  -0b "<< bitset<8>(c)<< endl ;
//Right Shift
c = a >> 2;         // 15 = 0000 1111
cout << "Line 6 - Value of c is: " << c
//Print register
<< "\n\t Right Shift register a>>2: -0b "<< bitset<8>(c)<< endl ;
return 0;
}

```

Assignment Routine

```
#include <iostream>
using namespace std;
int main() {
    // Assigning value 10 to a
    // using "=" operator
    int a = 10;
    cout<<"This program will atomically manipulate the Value of a"
    <<"\n\tThe value of a is " <<a;

    // Assigning value by adding 10 to a
    // using "+=" operator
    a += 10;
    cout<<"\n Atomically + 10: the value of a is now " <<a;

    // Assigning value by subtracting 10 from a
    // using "-=" operator
    a -= 10;
    cout<<"\n Atomically - 10 from a: the value of a is now " <<a;

    // Assigning value by multiplying 10 to a
    // using "*=" operator
    a *= 10;
    cout<<"\n Atomically * 10: the value of a is now " <<a;

    // Assigning value by dividing 10 from a
```

```
// using "/"= operator
a /= 10;
cout<<"\n Atomically dividing by 10: the value of a is now " <<a ;

return 0;
}
```

While Loop featuring a Unary Operator

```
#include <iostream>
using namespace std;
int main() {
    //Initialize the counter, i
    int i = 0;
    //While i is less than five, print i
    while (i < 5) {
        cout << i << "\n";
        i++;    //Unary counter i
    }
    return 0;
}
```

Unary Operator Routine with Post- and Pre Operations

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, buf;

    cout<<"This program will use unary operators to increment \nand
    decrement variable a and a buffer"

    <<"\n\tThe value of a is " <<a;

    // post-increment
    buf = a++;
    // a becomes 11 now
    cout << "\nPost-Increment: a is "<<a<<" and buf is "<<buf;

    // post-decrement example:
    buf = a--;
    // a becomes 10 now
    cout << "\nPost-Decrement:a is "<<a<<" and buf is "<<buf;

    // pre-increment example:
    // buf is assigned 11 now since a is updated here itself
    buf = ++a;
    // a and res have same values = 11
    cout << "\nPre-Increment:a is "<<a<<" and buf is "<<buf;

    // pre-decrement example:
```

```
// res is assigned 10 only since a is updated here itself
buf = --a;
// a and res have same values = 10
cout << "\nPre-Decrementt:a is "<<a<<" and buf is "<<buf;
return 0;
}
```

Conditional Operations Demo

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    cout << "This program will pick the greatest one
using two "
        << "\nmethods: a conditional operator and"
        << " an \nif()... else statement."
    << "\n\tWe expect a result of 5.";
    cout << "\n\nExecute expression using "
    << " ternary operator: ";
    // Execute expression using
    // ternary operator
    int a = 2 > 5 ? 2 : 5;
    cout << a << endl;

    cout << "\nExecute expression using "
    << "if else statement: ";

    // Execute expression using if else
    if ( 2 > 5)
        cout << "2";
    else
        cout << "5";
    return 0;
}
```


While Loop Demo

```
#include <iostream>
using namespace std;
int main() {
    //Initialize the counter, i
    int i = 1;
    //Print explanation of program
    cout <<"This program will number and print 10 lines-";
    //While i is less than 11, print "Hello World"
    while (i < 11) {
        cout <<"\n\t"<< i << " Hello World";
        i++;    //Unary counter i
    }
    return 0;
}
```

For Loop Demo

```
#include <iostream>
using namespace std;
int main() {
    //Print explanation of program
    cout <<"This program will number and print 10 lines-";
    //While i is less than 11, print "Hello World"
    for (int i= 1; i < 11; i++ ) {
        cout <<"\n\t"<< i << " Hello World";

    }

    return 0;
}
```

Do While Loop Demo exhibiting an exit controlled loop

```
#include <iostream>
using namespace std;
int main() {
    //Initialize the counter, i
    int i = 11;
    //Print explanation of program
    cout <<"This program may or may not print one
    \"Hello World\"";
    //While i is less than 11, print "Hello World"
    do {
        cout <<"\n\t"<< i << " Hello World";
        i++;    //Unary counter i

    }
    while (i < 11);

    return 0;
}
```

Nested If-Else Demo

```
#include <iostream>
using namespace std;
int main() {
    cout << "This program will go through a sequential"
    << "\nlist of numbers 2 through 4 and pick the "
    << "\ngreatest none using a nested if-else "
    << "\nstatement."
    << "\nGreatest number:\n\t";
    if ( 2 > 3 )
        cout << "2";
    else if ( 3 > 4 )
        cout << "3";
    else
        cout << "4";
    return 0;
}
```

Nested If-Else and Nested Ternary Operation Comparison

```
#include <iostream>
using namespace std;
int main() {
    cout << "This program will go through a sequential"
    << "\nlist of numbers 2 through 4 and pick the "
    << "\ngreatest none using two methods:"
    << "\na nested if-else and a nested ternary operator"
    << "\n Nested If-else Greatest number:\n\t";
    if ( 2 > 3 )
        cout << "2";
    else if ( 3 > 4 )
        cout << "3";

    else
        cout << "4";

    cout << "\n Nested ternary operator Greatest number:\n\t";
    int a = 2 > 3 ? 2 : 3 > 4 ? 3 : 4;
    cout << a;
    return 0;
}
```

Switch Demo

```
#include <iostream>
using namespace std;
int main() {
//Initialize variable
int x = 2;
//Print explanation
cout<< "This program prints a choice given x.\n"
<<"\n Variable x is equal to: "<< x
<<"\n\t";
// Program switch
    switch (x)
    {
        case 1:
            cout << "Choice is 1";
            break;
        case 2:
            cout << "Choice is 2";
            break;
        case 3:
            cout << "Choice is 3";
            break;
        default:
            cout << "Program Exit";
            break;
    }
return 0;
}
```

Function Demo

```
#include <iostream>
using namespace std;
// Function declaration
void myFunction();
// The main method
int main() {
    myFunction(); // call the function
    return 0;
}
// Function definition
void myFunction() {
    cout << "I just got executed!";
}
```

Functions with Parameter Passing

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

//Declare function
int max(int x, int y);
//Main
int main() {
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m;
    return 0;
}
//Define function
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```


Program without function overloading

```
#include <iostream>
using namespace std;
int plusFuncInt(int x, int y) {
    return x + y;
}
double plusFuncDouble(double x, double y) {
    return x + y;
}
int main() {
    int myNum1 = plusFuncInt(8, 5);
    double myNum2 = plusFuncDouble(4.3, 6.26);
    cout << "Int: " << myNum1 << "\n";
    cout << "Double: " << myNum2;
    return 0;
}
```

Program function overloading

```
#include <iostream>
using namespace std;
int plusFuncInt(int x, int y) {
    return x + y;
}
//Change function
double plusFuncInt(double x, double y) {
    return x + y;
}
int main() {
    int myNum1 = plusFuncInt(8, 5);

    //Call function in place of the old one
    double myNum2 = plusFuncInt(4.3, 6.26);
    cout << "Int: " << myNum1 << "\n";
    cout << "Double: " << myNum2;
    return 0;
}
```

References

- Running C,C++ Programs in Linux] Ubuntu 16.04 (Ubuntu Tutorial for Beginners). (n.d.). Retrieved from <https://www.youtube.com/watch?v=A7Ny-ro1hT0>
- 1134.00—Web Developers. (n.d.). Retrieved September 20, 2019, from <https://www.onetonline.org/link/summary/15-1134.00>
- 1199.03—Web Administrators. (n.d.). Retrieved October 1, 2019, from <https://www.onetonline.org/link/summary/15-1199.03>
- arwal, H. (2017a, January 13). Loops in C and C++. Retrieved August 28, 2019, from [GeeksforGeeks website: https://www.geeksforgeeks.org/loops-in-c-and-cpp/](https://www.geeksforgeeks.org/loops-in-c-and-cpp/)
- arwal, H. (2017b, May 16). Setting up C++ Development Environment. Retrieved August 13, 2019, from [GeeksforGeeks website: https://www.geeksforgeeks.org/setting-c-development-environment/](https://www.geeksforgeeks.org/setting-c-development-environment/)
- asthi, S. (n.d.). Switch Statement in C/C++—GeeksforGeeks. Retrieved October 3, 2019, from <https://www.geeksforgeeks.org/switch-statement-cc/>
- iefish Editor: Features. (n.d.). Retrieved August 28, 2019, from [Bluefish Editor: Features website: http://bluefish.openoffice.nl/features.html](http://bluefish.openoffice.nl/features.html)
- + | Nested Ternary Operator. (2018, October 15). Retrieved October 3, 2019, from [GeeksforGeeks website: https://www.geeksforgeeks.org/c-nested-ternary-operator/](https://www.geeksforgeeks.org/c-nested-ternary-operator/)
- + Code [Paste Site]. (2015, December 5). Retrieved August 28, 2019, from [Pastebin.com website: https://pastebin.com/f7KKzVyf](https://pastebin.com/f7KKzVyf)
- + Function Overloading. (n.d.). Retrieved October 4, 2019, from https://www.w3schools.com/cpp/cpp_function_overloading.asp
- + Functions. (n.d.). Retrieved October 3, 2019, from https://www.w3schools.com/cpp/cpp_functions.asp

- + Operators. (n.d.). Retrieved August 28, 2019, from https://www.w3schools.com/cpp/cpp_operators.asp
- + Variables. (n.d.). Retrieved August 28, 2019, from https://www.w3schools.com/cpp/cpp_variables.asp
- + While Loop. (n.d.). Retrieved October 3, 2019, from https://www.w3schools.com/cpp/cpp_while_loop.asp
- ntos-faq | Open Source Community. (n.d.). Retrieved September 30, 2019, from <https://community.redhat.com/centos-faq/>
- b, R. (n.d.). What is the Base-10 Number System? Retrieved October 2, 2019, from ThoughtCo website: <https://www.thoughtco.com/definition-of-base-10-2312365>
- escape sequences—Cplusplus.com. (n.d.). Retrieved October 1, 2019, from <https://en.cppreference.com/w/cpp/language/escape>
- C 7 Release Series—Changes, New Features, and Fixes—GNU Project—Free Software Foundation (FSF). (n.d.). Retrieved September 30, 2019, from <https://gcc.gnu.org/gcc-7/changes.html>
- How much data does YouTube store? - Quora. (n.d.). Retrieved October 4, 2019, from <https://www.quora.com/How-much-data-does-YouTube-store>
- Input/output manipulators—Cplusplus.com. (n.d.). Retrieved October 1, 2019, from <https://en.cppreference.com/w/cpp/io/manip>
- Introduction. (n.d.). Retrieved August 28, 2019, from <https://web.stanford.edu/class/cs98si/slides/overview.html>
- gati, A. (2015a, July 28). Operators in C | Set 1 (Arithmetic Operators). Retrieved August 28, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/operators-in-c-set-1-arithmetic-operators/>
- gati, A. (2015b, July 29). Operators in C | Set 2 (Relational and Logical Operators). Retrieved August 28, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/operators-in-c-set-2-relational-and-logical-operators/>
- liam, J. (2012). Understanding Bitwise Operators. Retrieved August 28, 2019, from <https://code.tutsplus.com/articles/understanding-bitwise->

operators--active-11301

- owska, K. (2019, April 23). What is C++ used for? 10 extremely powerful apps written in C++ - Blog. Retrieved August 28, 2019, from BoostHigh Software Development Company website: <https://boosthigh.com/10-extremely-powerful-apps-written-in-cpp/>
- mar, H. (2017, June 2). Unary operators in C/C++. Retrieved August 28, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/unary-operators-cc/>
- t of game engines. (2019). In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=List_of_game_engines&oldid=911528135
- hmee. (2018, October 18). What is the Difference Between Machine Code and Assembly Language. Retrieved September 11, 2019, from Pediaa.Com website: <https://pediaa.com/what-is-the-difference-between-machine-code-and-assembly-language/>
- nideep, P. (2017, October 26). Server side and Client side Programming. Retrieved September 30, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/server-side-client-side-programming/>
- erators-In-C.png (800×533). (n.d.). Retrieved August 28, 2019, from <https://www.geeksforgeeks.org/wp-content/uploads/Operators-In-C.png>
- n, A. (n.d.). Behind The App: The Story Of Notepad++ | Lifehacker Australia. Retrieved August 28, 2019, from <https://www.lifehacker.com.au/2015/06/behind-the-app-the-story-of-notepad/>
- el, Y. (2014). *How do I install gcc on Ubuntu Linux*. Retrieved from <https://www.youtube.com/watch?v=cotkJrewAz0>
- el, Y. (2017). *How to Create First C++ Hello World Project using Xcode Mac OS X*. Retrieved from https://www.youtube.com/watch?time_continue=21&v=-H_EyIqBNDA
- hev, M. (2017, November 3). As a programmer, is it important to have a GitHub profile? - Quora. Retrieved September 11, 2019, from <https://www.quora.com/As-a-programmer-is-it-important-to-have-a->

GitHub-profile

- bhu, R. (2018, December 11). Assignment Operators in C/C++. Retrieved August 28, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/assignment-operators-in-c-c/>
- programming software and the IDE - Revision 1—GCSE Computer Science. (n.d.). Retrieved September 11, 2019, from BBC Bitesize website: <https://www.bbc.co.uk/bitesize/guides/zgmpr82/revision/1>
- hon vs C++—Find Out The 9 Important Differences. (2018, July 1). Retrieved September 30, 2019, from EDUCBA website: <https://www.educba.com/python-vs-c-plus-plus/>
- What are Scripts? - Definition from Techopedia. (n.d.). Retrieved October 3, 2019, from Techopedia.com website: <https://www.techopedia.com/definition/10324/scripts>
- What is a Software Development Kit (SDK)? - Definition from Techopedia. (n.d.). Retrieved September 29, 2019, from Techopedia.com website: <https://www.techopedia.com/definition/3878/software-development-kit-sdk>
- What is Debugging? - Definition from Techopedia. (n.d.). Retrieved September 28, 2019, from Techopedia.com website: <https://www.techopedia.com/definition/16373/debugging>
- What's New in Xcode 9. (n.d.). Retrieved September 28, 2019, from https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/Xcode_9_release_notes/index.html
- eder, P., Butler, J. M., Theilmann, W., & Yahyapour, R. (2011). *Service Level Agreements for Cloud Computing*. Springer Science & Business Media.
- Writing first C++ program: Hello World example. (2017, May 17). Retrieved October 1, 2019, from GeeksforGeeks website: <https://www.geeksforgeeks.org/writing-first-c-program-hello-world-example/>
- Xcode. (n.d.). Retrieved September 29, 2019, from Mac App Store website: <https://apps.apple.com/us/app/xcode/id497799835?mt=12>