

```
void gotoxy(int column, int row)
{
    COORD coord;
    coord.X = column;
    coord.Y = row;
    SetConsoleCursorPosition(
        GetStdHandle(STD_OUTPUT_HANDLE),
        coord
    );
}

cout << "%d bytes from %s:" <<
    icmp_seq = %d. " << endl;
```



Understanding Programming: How to See the Unseen with C++

By Chad Jordan – May 12th, 2006

In this guide, you will learn:

- 1) Theoretical concepts behind the *what*, *how*, and *why* of computer programming
- 2) The hierarchical programming construct and systematic problem-solving
- 3) Visualizing methodologies, processes, and communication between data
- 4) Conditional statements, logical expressions, and selection control structures
- 5) Functions, data abstraction, classes, arrays, and recursion with C++

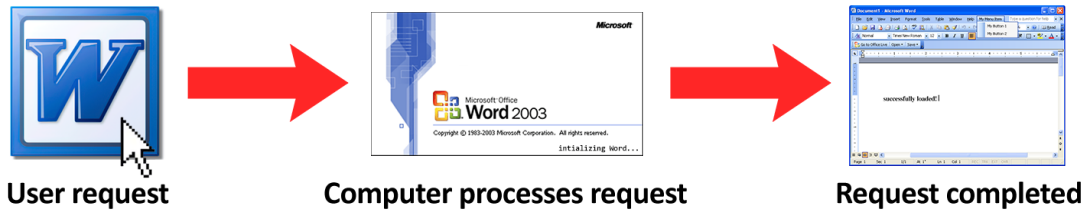
Introduction

In the 21st century, programming has remained an intimidating concept to many people. The reality is if you enjoy puzzles, are willing to work hard, you're patient, apply systematic problem-solving, and add a little creativity, programming can actually be really fun! As a puzzle enthusiast and computer science new media major, I can attest to the challenges that programming has presented to me. Full disclosure to inquiring minds, programming is hard! If it was easy, everyone would be doing it to get paid higher salaries. My professors used to say, *"Programming is the ability to see the unseen."* My hope is that this guide will help you see a glimpse into the universe of 1's and 0's. Computers only understand very specific rules and instructions in order to carry out the desired tasks from the developer/engineer. It's true that computers are incredible machines, capable of remarkable things, but engineers have to go through rigorous steps of communicating with the computer in order to successfully build and execute a program. For years society has believed that computers are these advanced machines intelligent enough to steal all of our jobs. The truth is, on their own computers are actually quite dumb. They have to be told every little detail in a very sequential, and many times tedious manner in order to properly compile information and run programs. They only perform and carry out the tasks that they do because we tell them how to do it. In hindsight, engineers are what bring the computer to life. The drawback of computer science is even if your code successfully compiles and runs, applications can still become unstable and crash without regard to reason. I've literally spent hours with the best professors debugging program errors only to finally throw our hands up in defeat and then I had to start programming all over again using a different approach. In mathematics, you are presented with a problem, and there is only one path to get the solution, but in computer science, the path veers off into numerous other possible directions in order to reach the successful execution of the compiler.

Just as in behavioral science with psychology we understand the human mind is one of the most complicated areas of study, computer science is the psychology of a computer and how it thinks, behaves, and allows us to communicate with it. My goal in this guide is to provide visual insight into learning how the psychology of the computer works and thus, how to write basic computer programs in C++. To do this, I will be providing code examples from computer programs that I've written using the free Dev C++ compiler in a Windows environment. This guide will be written with the understanding that the reader has had little to no experience in programming. I will be using an array of visual approaches to help teach some of the basic concepts and processes needed to start writing your own programs by the time you are done reading the guide. While my hope is that you will learn how to program using this guide, it's important to realize that I will by no means be covering everything you need to know to be a top programmer, in fact not even close. I'm covering what I believe are the essential topics to help you understand how it works so you can at least begin the journey. With that, let's begin!

What is a Computer?

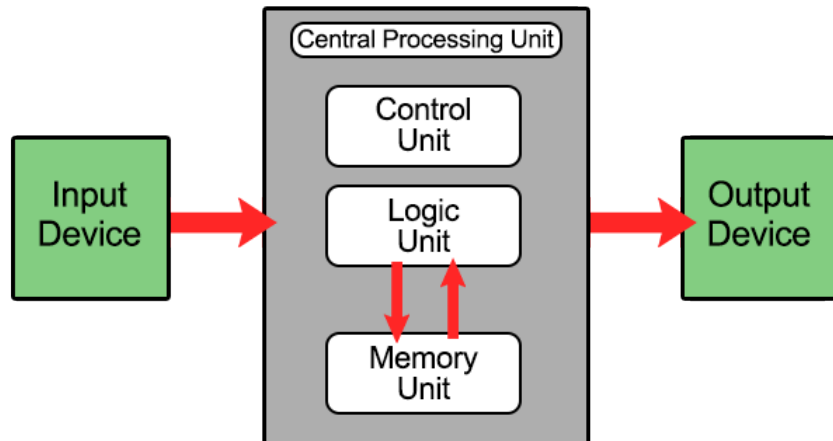
We start by looking at the computer. What is a computer, and what can we do with it? A lot of people use their computers in an office setting where they process Word documents, Excel sheets, or make PowerPoint presentations. Other people that work in financial businesses such as a bank or the stock market will use computers to run a series of calculations to roll out numerical results. No matter what you do, there is always a request (*initialization*) performed by the user, the computer takes the request to process it, and completes the requested task.



As the computer processes information and tasks, it's important to understand how these processes work. Speaking of how things work, there are a couple of facts to note for newcomers that are curious about this field:

- 1) You do not have to know much about computers to be a programmer, but it helps.
- 2) You do not have to be good at mathematics to be a programmer, but it helps.

Speaking for myself regarding these two statements, I have worked with numerous computers for several years now, so I've had time to learn about objects and data within different systems. However, I consider myself very average when it comes to mathematics. I've known multiple programmers who work full-time in the field, and math has never been their strong suit, nor something that they've enjoyed. What you really need is the ability to think systematically using problem-solving and logic. This is the biggest thing that will help you be a successful programmer. As I mentioned, I've worked with computers and researching systems for several years now, so let's take a look at how the computer will process this information.



At the starting point, we look at the **input device**. When we use keyboards to give instructions and commands such as code, this is the stage of the input. Next, it gets compiled through the **CPU** which takes the information and converts it to binary data. The CPU has two extra components, known as the **control unit**, and the **logic unit**. The **control unit** ensures the other data components are executed in proper sequential order. The **logic unit** performs the

arithmetic operations such as addition, subtraction, multiplication, and division. This also includes logic expressions on operands and variables. The **memory unit** assists the control and logic units in handling informational tasks. Once everything has been properly read by the compiler and passed through the CPU for conversion, it is then passed to the **output device**, also known as the video display/monitor. There's obviously a lot more involved with a computer than these simple procedures, but the above I/O diagram is the most important part when comprehending our statically-typed compiling (*console-based*) system for programming.

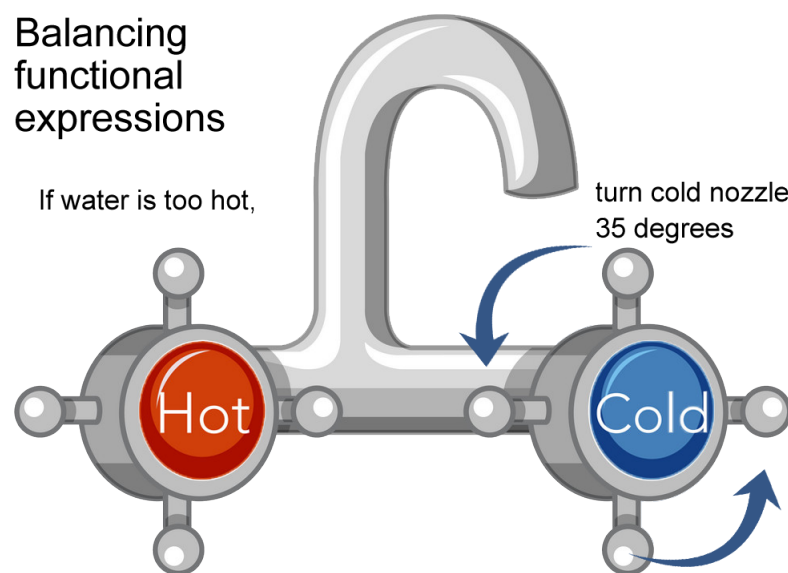
What is Programming?

As the user, you have various programming languages (*technologies*) at your disposal. Each of these technologies brings a vastly different learning curve of theoretical concepts, methodologies, procedures, and executions. No matter what kind of application you are trying to create, the purpose of programming is to create the perfect system. The programmer must thoroughly consider what kind of program they want to create, algorithmic procedures of what the application will require, ensure the implementation is clear with solid data flow, and above all else, how it needs to function for the end-user. A lot of people don't stop to think about how long we've had assistance from computers all around us. Without computers, there would be no day-to-day traffic direction, no air traffic control systems, no working machines in hospitals, no robotics of any kind, no electronic banking systems, and no modern telephones or cell phones would exist. However, even with all of these things being dependent on computers, without programming, computers would literally do nothing. Computer programming is the cornerstone of nearly all functional machines and electronics in the world.

In programming, there are different areas that focus specifically more on computers, then there are areas that focus more on machines, and programming languages for the web. Within these realms of technology exist what are known as high-level programming languages, and low-level programming languages. **High-level languages** are more programmer (*user*)-oriented, easier to understand, easier to debug, cross-platform, and more widely used. **Low-level languages** are more machine-oriented, harder to understand, more difficult to debug, maintenance is more complex, and requires an assembler to translate instructions. From this, we understand that different languages have different strengths for different platforms. The most common high-level languages in computer programming are C++, C#, Java, and Python. Then under the web, we have programming languages like Javascript, Ruby, PHP, Perl, and SQL mostly focusing on the back-end. Then we have the front-end, which primarily is coding for markup & styling such as HTML, XML, SGML, and CSS. We also have what is referred to as functional programming when dealing with focuses like artificial intelligence, machine learning, language processing, modeling vision, and speech. For those, we turn to languages like Haskell, Elixir, ML, and Lisp. For low-level programming, we've seen COBOL, Fortran, Machine Language, and Assembly (*8008, 8086, i386, i586, i686, Mano, x86_64 architectures*). The ocean of programming languages with their meanings and purposes is vast and deep. The question is, what does it take to get to a place of understanding these technologies and how to implement them? This question goes back to my statement made in the introduction about applying systematic problem-solving. Well, let's take a look at what you have to understand when moving forward in this field. Every line of code or command that you place, means something important.

E.g. I want to write a happy greeting to my computer like, “*Good morning computer! Isn’t it a wonderful day?*” Or, maybe I want to input a simple math problem like 3×4 and get the response, so I explain to the computer how to understand basic arithmetic operations. Everything in information systems has an **input** and an **output** no matter what kind of form is being used. To read this input and output, computers read things just as humans do, from top to bottom, and left to right. We always understand this simple step-by-step procedure, and so does the machine. It just happens to read it in an artificial language that it understands. Just as humans understand natural languages like English, Spanish, French, German, Russian, Japanese, etc. computers understand artificial languages like C, C++, C#, Java, Python, Assembly, etc. so we just have to learn these languages so that we can properly communicate with the computer. As humans within our own languages, we understand nouns, pronouns, adverbs, adjectives, and prepositions. When it comes to computers, they understand data types, variables, operands, conditionals, and classes. There are rules when following proper grammar when we write out everyday sentences to people in emails or other letters, and the same thing applies to artificial languages. There are rules of how things have to be presented to the computer in a sequential manner so that it understands what you’re wanting to do. This base level of understanding communication between humans, when compared to communication between data, has really been a method that has helped me, and if you think of it in this manner, it’s obviously different, but in many ways, it’s also very similar.

I mentioned input and output earlier, along with conditionals. This is very important for us to grasp when looking at the *programming thought process*. Think about the process of real-world tasks like the process of filling up a bathtub with water.

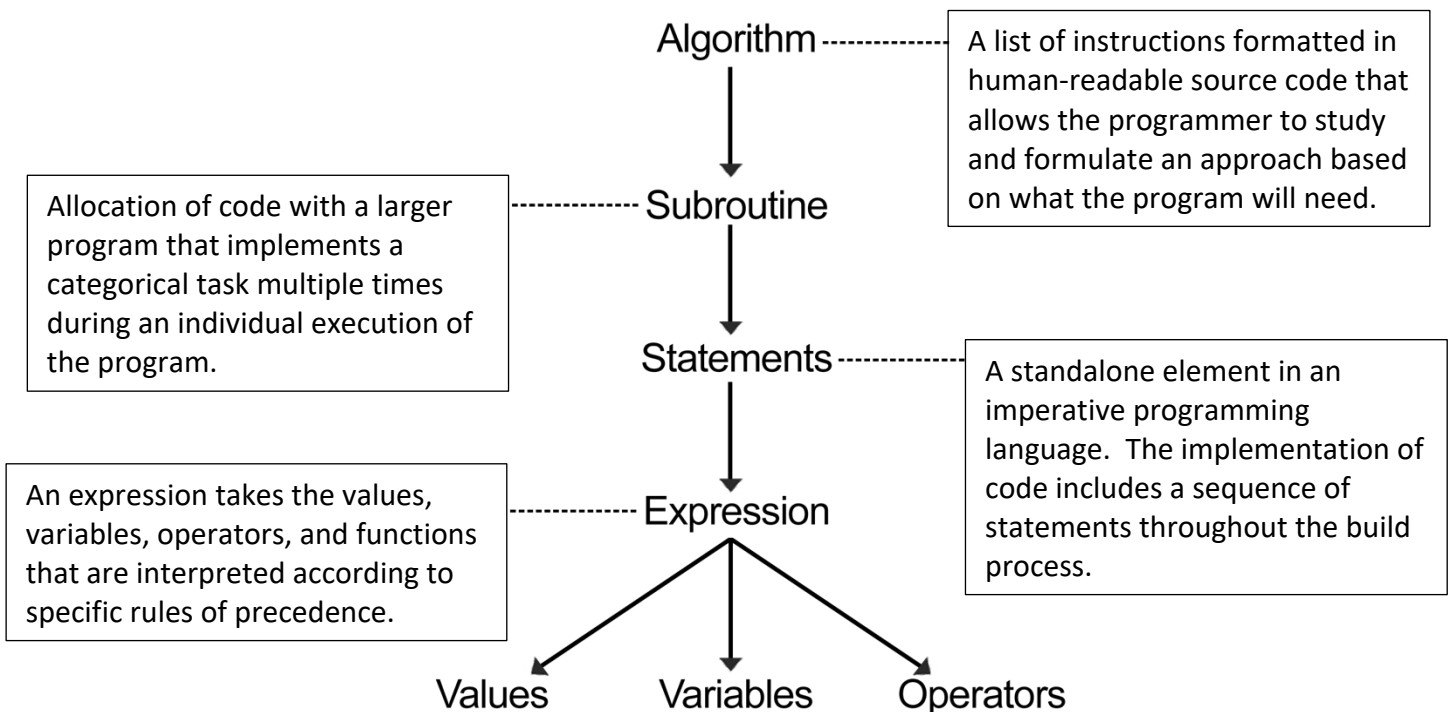


Everything around us is based on conditions. We perform the actions until we are pleased with the results. We perform the same actions in programming. *If this, \geq then this other thing, do this, otherwise, do that. If this is not found, keep searching for the thing until it’s found.* Yes, those are rather vague concepts, but they are the basis of sequential thinking in the computer science process. Everything tends to branch out in a hierarchical manner when met with the

conditional thought process. When thinking about conditional statements with input and output, we know that the computer waits for a button to be pressed. The same is done in our code. We can write a condition that checks for a specific button to be pressed, and if it's pressed, another part of the program is executed. If not, then the program continues exactly as specified until another condition is met. It should be thoroughly understood that while computer science is very objective (*right or wrong*), it is not as black and white as people may think. There are plenty of complexities (*gray areas*) that must be taken into consideration for good programming to be properly executed. Instructions must be given so the computer can carry out the tasks, and these instructions come in the form of writing functions, conditional loops, and other commands so that the request is put in, the computer processes the request, and then the request is completed. Based on this knowledge, we know there is a sequence and structure to programming. Another programming concept that my professors told me in the beginning, the key to understanding programming is:

- 1) **Sequence**
- 2) **Selection**
- 3) **Repetition**

You may be thinking, that those steps seem very vague, and people need more than Sequence, Selection, and Repetition in order to write computer programs. These steps are obviously a part of a much larger picture, it's more about allowing yourself to begin the mental process of understanding the structured sequence of programming. With these basic principles in mind, let's unfold the bigger picture and look at the **hierarchical programming construct**.



In computer programming, a **value** is a sequence of bits that is interpreted according to the data type that has inherited it such as integers, floating points, or strings. A **variable** is an identifier that is connected to a value and saved in system memory as an expression to be executed later. An **operator** is similar to what is used in mathematics such as +, -, *, / or = and

is typically a fixed number of built-in operators. This form of sequential thinking gives rise to the systematic thought process. When we use problem-solving, we assess the situation using a sequential thought process of executions to address each task until the problem or current stage of the problem is solved. **Systematic problem-solving** isn't much different.

- Recognize that a problem exists
- Analyze the problem
- Identify possible causes/solutions to the problem
- Evaluate the possible causes/solutions
- Develop an action to correct the problem
- Verify that the problem has been resolved

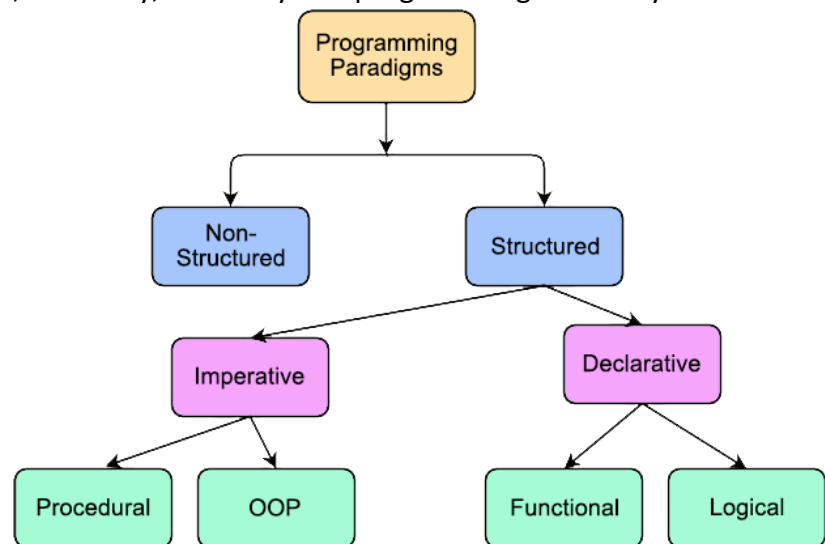
It is this mentality and process that you need to repeatedly focus your efforts on. Allow your mind to open up to these procedures and adapt your way of thinking to them. Programming aside, if more people applied these types of steps to their daily lives, we would have a lot more functional minds having the desire to work together. Now that you understand more about what programming is, the next question is understanding the *how* and the *why* behind programming with the types of styles (*paradigms*) of programming languages.

Programming Paradigms

We all understand that computers perform a vast number of tasks for us, but how many have stopped to think about how we can truly communicate with them? In a way, we communicate with them by putting in simple day-to-day requests such as running software applications. Likewise, when it comes to programming, the principles of what we're following are very similar to my above example with initializing a Microsoft word application. The user puts in a request, the request gets processed, and if all goes well, we have a successful compilation/execution of our program. So how do we do this? How can we begin to visualize the steps involved with writing our own computer programs, and lastly, which style of programming best fits your thought process?

We know that programming has different styles or *paradigms*. It is never a single, static approach, but can be executed using multiple styles and methodologies. More to this topic, we have different paradigms to consider such as **imperative** which includes procedural, and object-oriented programming. These two styles of programming are by far the most

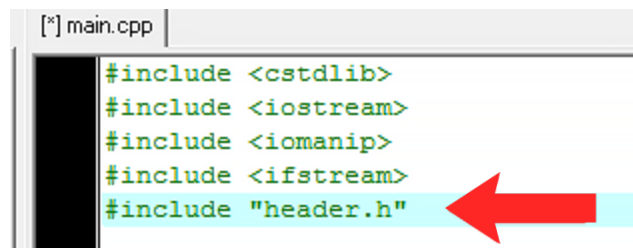
popular in computer science. Imperative consists of an array of instructions that are given to the computer to be executed in sequential order. Another paradigm is **declarative** which



includes functional, and logical programming. This style is less about providing instructions about how the computer should execute a task and more about what kind of result is required. This is one of the reasons why it's helpful to know a little about how computers work. The more you understand the processes of how data works and is executed, the better equipped you can be to decide which programming language you should choose for a given task or project. I will further address the topic of programming languages and their purposes shortly. Paradigms help to reduce complexity in programming. Although, they should never be confused with Languages! A paradigm is a style, a technique, an ideal method, intended to be implemented, it is not a programming language.

Learning About Libraries

In order for our programs to properly compile, the compiler has to understand the protocols in order to process the code that is being fed to it. To allow this we provide reference points (*also referred to as libraries*) at the beginning of our programs to provide the compiler with large lists of information and data so that it understands how to process our requests. Two of the C libraries that nearly every program requires in order to successfully run are called **iostream** and **cstdlib**. we define them as **#include <iostream>**, and **#include <cstdlib>**. IOstream essentially provides functionality to use an abstraction called *streams* designed to perform input and output operations. Cstdlib is a C-standard library that defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting. We include all various types of libraries in our programs, it just depends on what kind of programs we're writing. Take for example if we're wanting to generate an output file at runtime, we would use **#include <ofstream>**, and if we want to input information from an external file, we would use **#include <ifstream>**. File streaming processes and other data abstraction methods in programming are largely used in object-oriented programming. Another instance of input and output but with more details regarding data manipulation is the **<iomanip>** library. IOmanip is a list of helper functions to control the format of input and output. Similar to the logic unit of a CPU, the compiler understands basic arithmetic such as addition, subtraction, multiplication, and division. However, when it comes to more advanced numerical operations in C languages, we include the **<cmath>** numeric library. This library can assist us in retrieving square root results, decimal values, calculating logarithmic equations, sin/cos formulas in trigonometry, and a lot more. Even with the vast list of system libraries at our disposal, what happens when we want to write, customize, and reference our own library files? We can do exactly that! These are called header files so rather than ending with the file



```
[*] main.cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <ifstream>
#include "header.h"
```

extension, **.cpp** for a C++ file, they will end with a **.h** for *header* file instead. It's important to note that we call these two types of library files using two different methods. One is using angle brackets **< >**, as I've demonstrated above and to the right in this screenshot.

This means we are referencing pre-existing system libraries. The other method is with quotes: **#include "header.h"** and this tells the compiler that the file we are wanting to include is in a

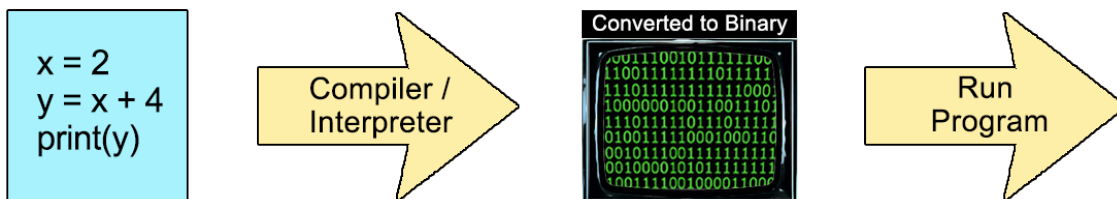
local directory/location. If my header file is located in another subdirectory away from my .cpp file, I would simply define it with the name of the created directory in front of the header name. E.g. `#include "some_directory/header.h"` In the earlier stages of programming, you're not in a place yet where you need to be concerned with creating multiple directories for handling input and output of data files. In the learning stages of programming, you can leave everything in one directory. C++ has extended libraries access, particularly, the popular Standard Template Library (STL). It helps to deploy data structures and many essential functions such as lists, stacks, vectors, and arrays in code to write it faster. For example, different containers are included, such as tables, queues, stacks, maps, and sets.

Visualizing Variables

Just like libraries, variables are used in every program you write. However, unlike libraries, we don't simply declare them once and then not reference them again. Variables are created and used in every aspect throughout your programs. We create them, we set values to them, and we reference them in our functions. So, let's visualize how we create and use variables. In C++, **variables** are containers that store data values such as **ints** (*integers*), **chars** (*characters*), **floats** (*floating-point numbers*), **bools** (*booleans*), and **strings**. In my home, I have containers that I use to store my computer components and peripherals. I always organize them by *type* so I use an electronic label maker to *label* the various containers with the type of hardware that corresponds to the label.



As such, we do the same thing virtually when writing our programs. Variables are just containers for storing data values. Understanding variables and how we store them is no different from applying basic arithmetic methods (E.g. $x = 2$ therefore, $4 + x = 6$). When we set 2 to x we are labeling the variable x with a value or amount of however many we wish when it comes to building our program the way we want. When we do this, we can organize any specific number of objects or lists that we wish to sort and print. The following diagram is a slightly altered, but similar way of looking at how we can set variables, calculate them, and the result is processed and then printed out to the screen.



Just these examples alone with the simplest form of arithmetic can help to paint a picture on a canvas for a clearer understanding of how we declare and use our data with variable values.

Syntax and Semantics of Code

Now for the real fun! We look at how the code is written and the meanings behind the various symbols, characters, and data types. We know that the human mind is a neural-network comprised of a vast system of impulses that are all running on a fast-paced, interconnected highway of signals. Computer programming is the artificial version of the human mind, especially when writing larger-scale programs with thousands of lines of code. C++ is one of the most commonly used, and fastest compiling languages for software development in the industry, but that doesn't mean it's the only option. Python is an interpretive programming language, not nearly as fast, but more versatile overall when it comes to programming environments, multi-purpose, and multi-platform applications.

What is syntax? **Syntax** is a particular way that you need to write and structure your code in a language so that the computer can read and execute it. **Semantics** are the *how* and the *why* behind the symbols, characters, and words in the code. Let's look at a simple 'Hello World' program between the syntax of C++ and Python.

C++:

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

int main()
{
    cout << "Hello World!" << endl;

    return 0;
}
```

Output: Hello World!

Python:

```
print ("Hello World!")
```

Output: Hello World!

The difference in the syntax is pretty apparent. Writing simple output code in Python is exponentially easier and faster to write than in C++. However, when it comes down to processing speed, C++ is much faster than Python. One of the biggest reasons for this is that C++ is a compiling language, whereas Python is an interpreted language. Interpreting going line by line through a program and immediately converting it to machine instructions. Much like an interpreter that listens and translates one incoming language to another, the programming language will take one command at a time and translate it to machine instructions and let it run. On the other hand, a compiler reads in the entire program all at once to determine what commands should be. The most important thing to remember no matter what language you use is that we are taking the code and converting it to machine commands.

Let's talk about the semantics of what's going on in this simple program. As we've already seen with Python, there isn't much of anything going on code-wise except using a print function to pass a string of, Hello World! In comparison to C++, the syntax is obviously not as straightforward. As we've learned earlier from the library section of this guide, we begin all of our programs by declaring the proper (*and standard*) libraries at the top.

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

On the next line we type, *using namespace std;* because when we run a program to print output, 'using namespace std' is saying if you find something that is not declared in the current scope go to this location and check **std** (*standard characters and names*) such as *cout*, *cin*, and *endl*. The semicolon that follows is simply an indicator that lets the compiler know that it's reached the end of the command, separating the command from the rest of the respective code. The semicolon like any other symbol in code is always required in order for the program to properly run.

Starting on the next line we write the main function because we need to return an integer before closing out of the program. When we declare a function, the format is:

data type, name of function, followed by a set of parentheses (**()**). The parentheses behave like

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

int main()
```

an expression similar to mathematics: $(a + b) * c$ and declares an argument and passes it in a function declaration. We don't always have to pass a parameter to the function, and in this instance, we simply declare it empty as two sets of (). In programming, the order of parentheses and brackets is executed

as (), {}, and []. As an example, if we execute a string, a valid string would be: "(() {})" and an invalid string would be "{ [] }". In more intermediate levels of programming, we use what are called stacks to traverse through the expression until it has been exhausted, and I will provide more code examples of how this is done later. I've provided this example of the order of parentheses because this is exactly the next step we perform in our simple output program. The opening and closing of the curly brackets specify that everything inside of the brackets such as commands, conditional statements, nested functions, strings, etc. remain between these two brackets. This means that the placement of the top curly bracket does not technically have to remain below the function, but can be declared either way:

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

int main()
{
}
}
```

(Better for readability)

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

int main() {
}
}
```

(Saves on black space)

Our next step involves putting in the appropriate output command inside of the curly brackets

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

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

for our main function. The *cout* command is a C-language output command that is outputting a command from the keyboard.

In the case of our Hello World! program, we are wanting to output a message (*string*) to the screen. To do

this properly for readability in our code, we always indent our command inside of the curly brackets when they are called. After typing

cout, the output command requires two << (*left-angle brackets or left shift operator*) which means 'put to' then we type two "" quotes because the quotes indicate to the compiler what we are wanting to pass to our output string. Put bluntly, or more in English, we are placing the text inside of the quotes which represent our message. Afterward, we type two more left-angle brackets followed by endl; in order to 'put-to' the end of the line endl and our semicolon to indicate the end of the command. As you progress through programming, you will eventually run into the cin command. Just as there is a cout command for output, there is a cin command for coding input operations. Using the >> (*right shift operator*) we can 'get-from' file data.

We're nearly done going over the semantics of a simple *Hello World!* program, but there's still one more instance that we need to cover. As I mentioned earlier, we have a function declaration (*int main*), and being an *int* datatype this requires returning an integer value before the program is completed. Since this integer is part of our main function, we expand the closing } curly bracket to allow space for one more command. We know that we have everything else in place when it comes to accounting for the output of a simple *Hello World!* message to the screen.

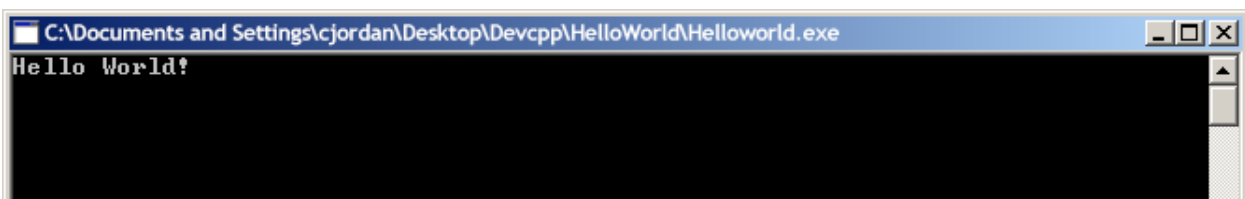
```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;

    return 0;
}
```

Our *int* function must return a value of either **1**, or **0**, (*True, or False*). This means if we return a value of true (*1*), then the program has not been executed successfully, and if we return false (*0*), the program has been executed successfully and therefore completed the intended request of the user.

When compiling and running the code, the user gets the following text in a console window:



The source code that I demonstrated above is one of the common methods to write the infamous *Hello World!* program. However, as I mentioned at the beginning of this guide, programming can be approached and written in multiple ways. Here are some more methods for writing a *Hello World!* program in C++ that will give us the exact same output:

```
#include <iostream>

int main() {
    puts("Hello World!");

    return false;
}
```

```
#include <iostream>

int main() {
    std::cout << "Hello
World!";

    return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <stdio.h>

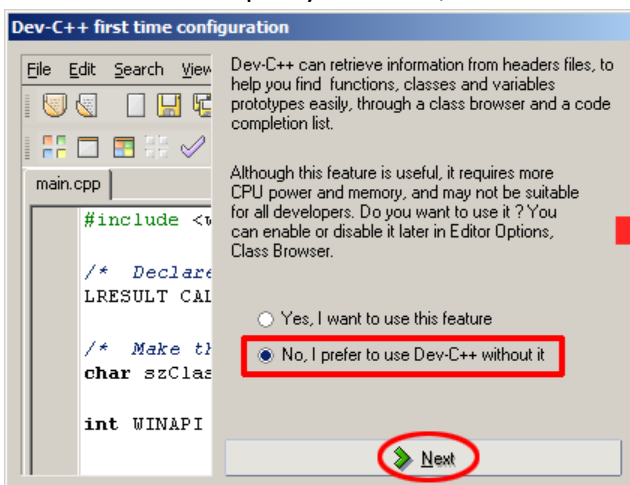
int main(void) {
    printf ("Hello
World!");
}
```

There are still many other ways to write it, but it's important to keep your code clean, and not go overboard. These three additional examples happen to be simple, and syntactically clean. As far as writing a program in C++, what you see here is as easy as it gets. If you would like to try writing your own *Hello World!* program, here are the steps to setting up Dev C++ on your Windows PC.

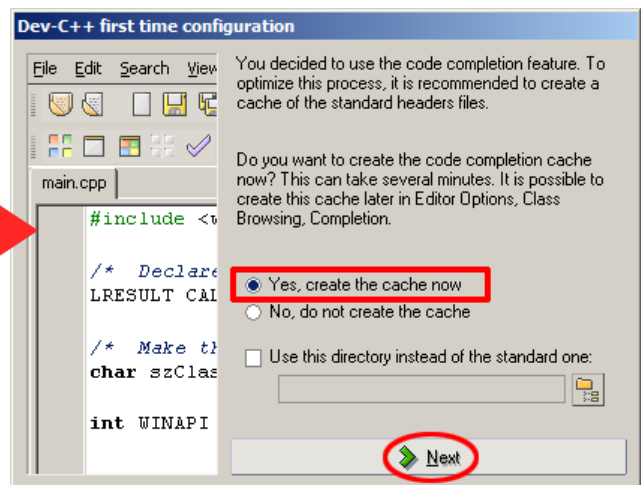
Installing Dev C++ on Windows

Begin by downloading and installing Dev C++ on your computer by clicking the following [link](#). Once downloaded, simply run the EXE file and go through the prompts confirming, agreeing, and hitting *Next* leaving everything as default throughout the process, click *Install*. This first part is so quick there's no need to provide screenshots. When you get to the end, **uncheck Run Dev C++ 4.9.9.2** because we have some quick configurations to set up first.

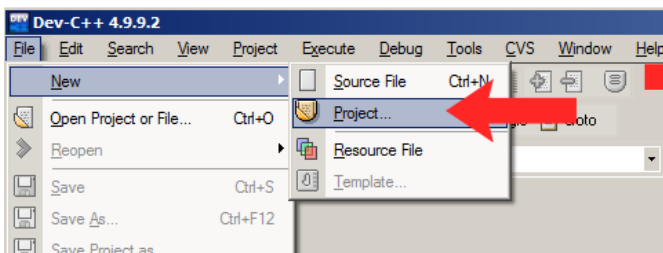
Configuration Step 1: Once you reach the first time configuration part of the installation, I clicked the **No** radio button because I don't care for the auto-complete and other unnecessary features. This is up to you. Then, hit *Next*.



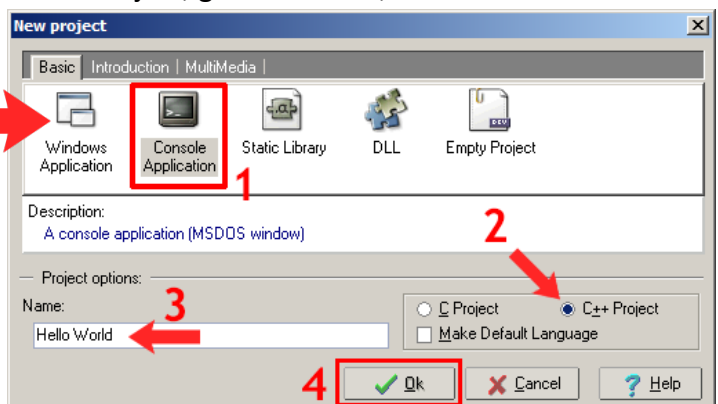
Configuration Step 2: On the next dialog box click the **YES** radio button to create cache now. Then, hit *Next*.



Configuration Step 3: After it's done parsing the files, you can complete any other clicks to get through the configuration, open the program, and click **File > New > Project**.

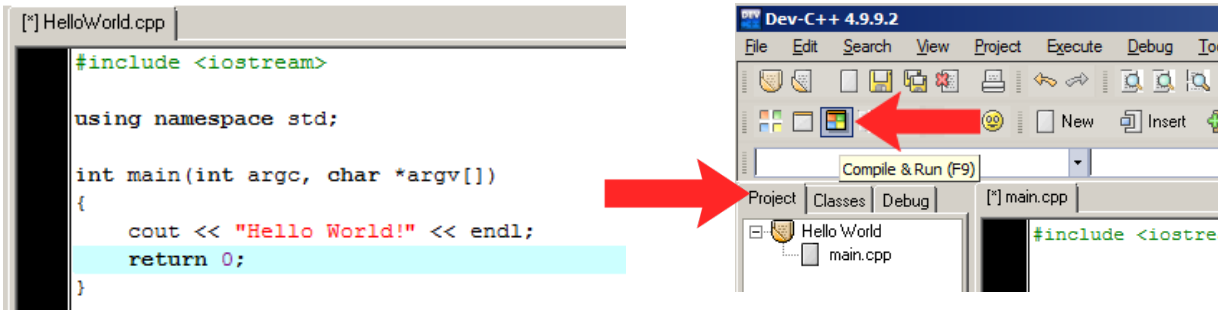


Configuration Step 4: Now we set up our Project as a **Console Application**, ensure it's a C++ Project, give it a name, and click **Ok**.

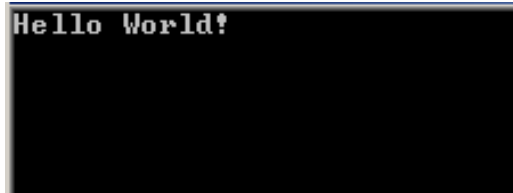


Ensure that your source code is set up to output *Hello World!* to the screen.

This button compiles and runs the code, and as you can see, you can also use **F9** as a keyboard shortcut.

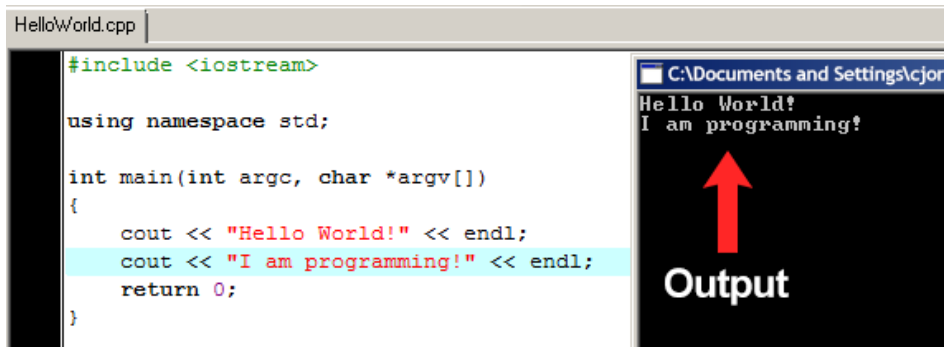


The result is a successfully compiled and ran *Hello World!* program.

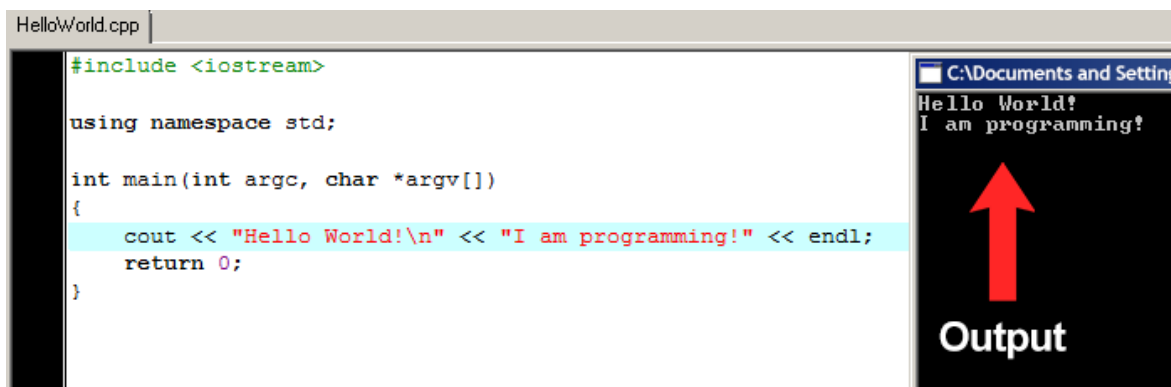


Now that you have at least one compiler installed on your computer, a general understanding of how to use Dev C++, and the basics of printing out a simple *Hello World!* message, let's kick it up a notch with formatting simple lists of strings.

We've printed out a single line, but what about printing out a new line on multiple lines? This can be accomplished by putting in another `cout` statement.



We can also enter line breaks in the strings which is another way to create a new line for our statements. This saves on space by including our next string in our original `cout` statement simply by adding another `<<` (*left shift operator*) to give us the exact same output.



While we can print out some simple phrases using hardcoded methods like cout, we can also use the `<string>` library to allow us the ability to work with string data types. String data types are used to store sequences of characters to print out sets of phrases and form them into sentences. This is also referred to as **string concatenation**.

```

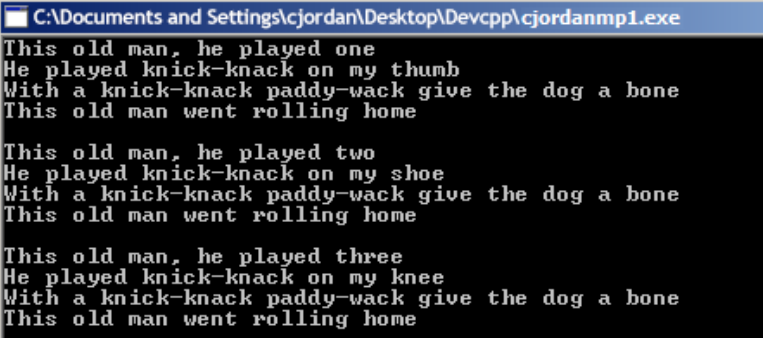
[*] Mp1.cpp
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
    const char SPACE = ' ';
    string line1 = "This old man, he played ";
    string line2 = "He played knick-knack ";
    string word1 = "one";
    string word2 = "two";
    string word3 = "three";
    string thumb = "on my thumb";
    string shoe = "on my shoe";
    string knee = "on my knee";

    cout << line1 << word1 << endl;
    cout << line2 << thumb << endl;
    cout << "With a knick-knack paddy-wack give the dog a bone" << endl;
    cout << "This old man went rolling home\n" << endl;
    cout << line1 << word2 << endl;
    cout << line2 << shoe << endl;
    cout << "With a knick-knack paddy-wack give the dog a bone" << endl;
    cout << "This old man went rolling home\n" << endl;
    cout << line1 << word3 << endl;
    cout << line2 << knee << endl;
    cout << "With a knick-knack paddy-wack give the dog a bone" << endl;
    cout << "This old man went rolling home\n" << endl;

    return 0;
}

```



Programming offers infinitely more than just printing out silly messages to a console application. We can start by doing some basic arithmetic operations very similar to the examples I provided on page 8 of my *'Visualizing Variables'* section.

```

#include <cstdlib>
#include <iostream>

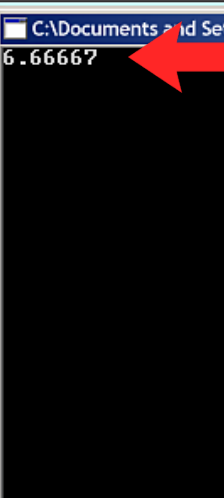
using namespace std;

int main(int argc, char *argv[])
{
    int num1 = 5;
    int num2 = 7;
    int num3 = 8;
    float sum;

    sum = num1 + num2 + num3;
    cout << sum/3;

    return EXIT_SUCCESS;
}

```

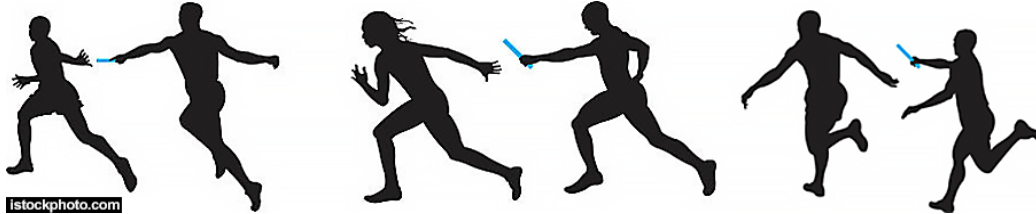


This is the correct output given the arithmetic operation:
 $5 + 7 + 8 / 3$

In this program, I declare three integer variables, assign three random values of 5, 7, and 8 to the variables, declare a float variable to hold the sum of the three values, calculate the average of the three integers, assign that value to the float variable, and output the float variable contents to the screen.

Conditionals and Expressions

When we understand the order in which statements are executed in a program, we call this the *flow of control*. As such, the computer is under control of one statement at a time. When a statement is executed, the control is turned over to the next statement (much like a baton being passed in a relay race).



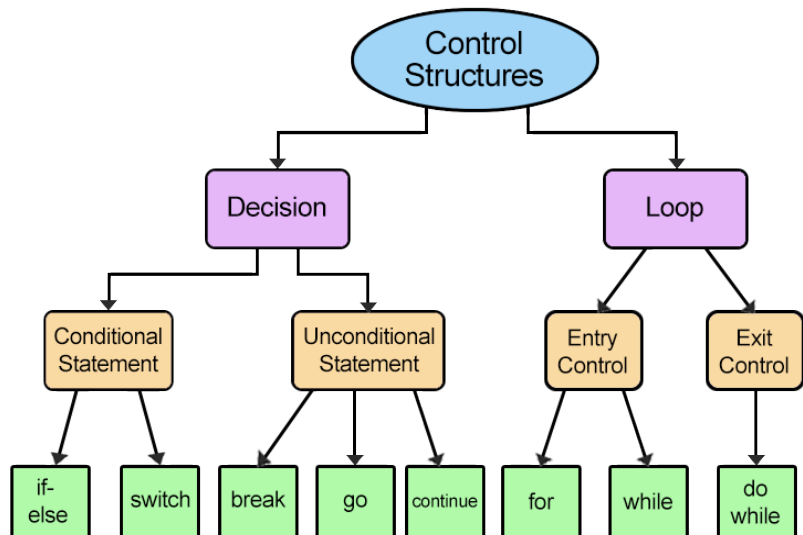
In programming languages, assertions take the form of logical expressions *also referred to as Boolean expressions*. Just as an arithmetic expression is made up of numeric values and operations, a logical expression is made up of logical values and operations.

When considering the flow of control, we realize any given function is implemented with three basic types of control structures:

Sequential – execution of code statements from top to bottom (one line after another).

Selection – choosing between two or more alternative paths (*if, if/else, switch*).

Repetition – used for looping (*for, while, do/while*) i.e. repeating a block of code multiple times in a row.



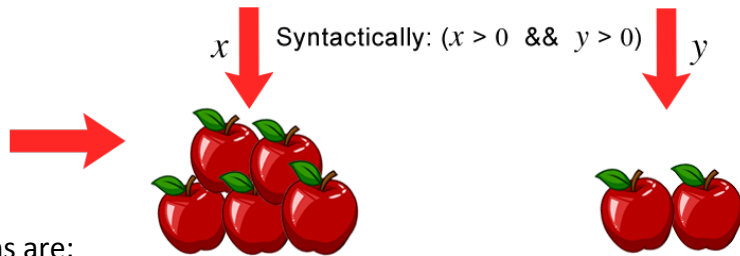
When using these conditions, we use logical operators to check whether an expression is true or false. **Logical operators** are arithmetic comparison operators much like the symbols we use in mathematics. Each of the following operators returns either true or false (1 or 0) and are used as test expressions in selection statements and or repetition (*loop*) statements.

```
x == y // x is equal to y
x != y // x is not equal to y
x < y // x is less than y
x <= y // x is less than or equal to y
x > y // x is greater than y
x >= y // x is greater than or equal to y
```

Booleans work the same way, returning a value of either true or false.

```
x && y // the AND operator -- true if both x and y are true
x || y // the OR operator -- true if either x or y (or both) are true
!x // the NOT operator (negation) -- true if x is false
```


In human terms, saying “*x* and *y*”
is like coding: “*x* && *y*”



Some examples of these expressions are:

```
(x > 0 && y > 0 && z > 0) // all three of (x, y, z) are positive
(x < 0 || y < 0 || z < 0) // at least one of the three variables is
negative

// there are at least 20 students and the class average is at least 70
(numStudents >= 20 && !(classAvg < 70))

// means the same thing as the previous expression
(numStudents >= 20 && classAvg >= 70)
```

The && and || operators also have a feature known as **short-circuit evaluation**. In the Boolean AND expression (*X* && *Y*), if *X* is false, there is no need to evaluate *Y* (so the evaluation stops).

Example:

```
(d != 0 && n / d > 0)
```

notice that the short circuit is crucial in this one. If *d* is 0, then evaluating (*n* / *d*) would result in division by 0 (illegal). The "short-circuit" prevents it in this case. If *d* is 0, the first operand (*d* != 0) is false. So the whole && is false.

Similarly, for the Boolean OR operation (*X* || *Y*), if the first part is true, the whole thing is true, so there is no need to continue the evaluation. **The computer only evaluates as much of the expression as it needs.** This can allow the programmer to write faster-executing code.

Technically, the C++ operators: **!**, **&&**, and **||** are not required to have logical expressions as operands. Their operands can be of any simple data type, even floating-point types.

The *if/else* selection statement is the most common selection statement in programming. The basic syntax would be:

```
if (expression)
    statement
else
    statement
```

The *else* clause is optional, so this format is also valid:

```
if (expression)
    statement
```

The expression part can be any expression that evaluates a value (an R-value), and it must be enclosed in parentheses (**()**). The best use is to make the expression a **Boolean expression**, which is an operation that always returns a value of **true** or **false**.

For other expressions (like $x + y$), for instance):

- an expression that evaluates to 0 is considered false
- an expression that evaluates to anything else (non-zero) is considered true

Example:

```
if (grade >= 68)
    cout << "Passing";
```

```
if (x == 0)
    cout << "Nothing here";
else
    cout << "There is a value";
```

This example sums the numbers starting with 1, through whatever stopping value of the user's choice using a count-controlled while loop and then printing out the total:

```
int main()
{
    int choice;
    int numCount;
    int total;
    cout << "How many numbers do you want to print out ";
    cin >> choice;
    total = 0;
    numCount = 1;
    while (numCount <= choice)
    {
        total = total + numCount;
        numCount++;
    }
    cout << total << endl;

    return 0;
}
```

Similar to this approach, we can also use a for loop to count and specify starting and ending points for printing out all of the numbers in between the user-specified values:

```
int main()
{
    int startNum, endNum;
    cout << "Enter the starting number: " << endl;
    cin >> startNum;
    cout << "Enter the last number: " << endl;
    cin >> lastNum;
    for (int start = startNum; start <= lastNum; start++)
    {
        cout << start << ", ";
    }
    return 0;
}
```

Arrays

Through several code examples, we've seen how to assign and store values of different data types with variables. Arrays are used to store multiple values within a single variable, instead of declaring multiple variables for each value. For example, this means that five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

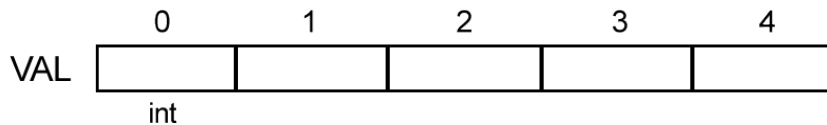
To declare an array, you define the variable type, specify the name of the array, followed by square brackets and specify the number of values within the brackets. The declaration would be `dataType arrayName[arraySize];`

Example:

`int val[5];`

- `int` – Type of element to be stored (*datatype*)
- `val` – Name of the array (*arrayName*)
- `5` – Size of the array (*arraySize*)

This example allows me to have 5 containers for storing values in an array. As we remember from storing values inside of variables, I can store however many values that I wish in each of the 5 containers.



In the above example, each blank panel represents an element of the array. In this case, these are integer data types. These elements are numbered from 0 to 4, with 0 being the first and 4 being the last. In C++, the first element in an array is always numbered with a zero, no matter its length. Therefore, the size of our array is a total count of 5.

Let's look at an example of a sequential search in a sorted list of an array.

```
void SearchOrd(int list[ ], int item, int length, int& index, bool& found)
{
    index = 0;
    list[length] = item;
    while (item > list[index])
    {
        index ++;
    }
    found = (index < length && item == list[index]);
}
```

In this example of a sorted list, the user can enter a sequence of numbers (*values*) into an array, and then the index starts at zero and searches through the array returning true until it hits 8, then drops out of the loop.

	item	list	length
	89	0	2
		1	7
		2	12
	0	3	13
		4	15
		5	29
		6	37
		7	55
		8	89
	found		
	False		

You just saw from my previous example how we can run a sequential search inside of an array, but how about inserting and deleting in an ordered list? We can do both of those!

To insert:

```
void Insert(int list[ ], int& length, int item)
{
    bool placeFound;
    int index;
    int count;
    SearchOrd(list, item, length, index, placeFound);
    for (count = length - 1; count >= index; count--)
    {
        list[count + 1] = list[count];
    }
    list[index] = item;
    length++;
}
```

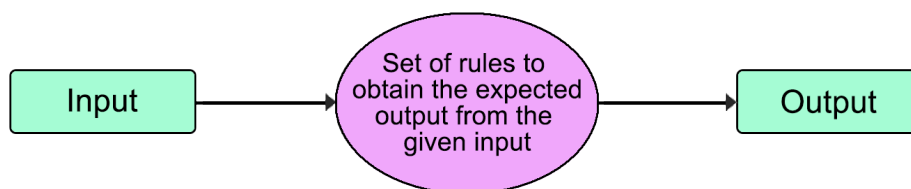
and to delete:

```
for (count = index; count <length-1; count++;)
{
    list[count] = list[count + 1];
}
length--;
```

Now, suppose we want to perform a selection sort in an array. We need to properly plan for this process, and to do so we create an algorithm. I will cover this process shortly after going over more about the concepts of algorithms.

What are Algorithms?

Continuing on with the theoretical side of computing, algorithms are a sequence of instructions that are applied to problem-solving operations. They can either be straightforward or very complex it simply depends on what you're trying to build.

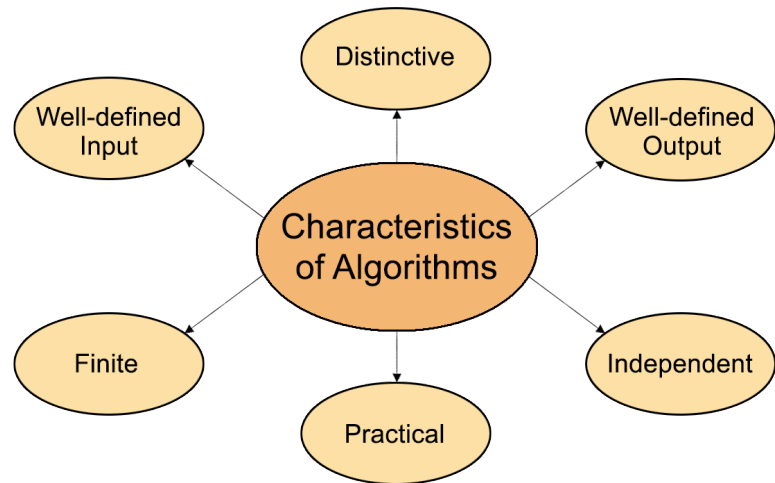


Speaking of instructions and building, applying algorithms is a lot like building a small piece of furniture with a list of instructions. When you purchase a small table or bookshelf from IKEA, it usually comes complete (*most of the time*) in the box with instructions. As we remove the parts and the instructions, we account for everything we know that we need to complete the build. Next, we consider how we have to put everything together. Do we have the proper tools, and knowledge of how to use said tools? If so, we can move forward to the instructions. Reading through the instructions is no different than what you are creating for your own computer program. You have a desired outcome for a functional piece, you ensure that you have a

sequential list of instructions (*operations*) in mind, and you simply build (*code*) by connecting one piece (*function*) at a time. Then, the final result (*output*) is the completed furniture. With real-world situations in mind, another way of looking at this is that you're technically using algorithms every time you use your computer, a calculator, or an ATM, you're using algorithms. As such, these simple, sequential processes of instructions (*algorithms*) help us perform tasks in programming.

In order for some instructions to be classified as an algorithm, it needs to have the following characteristics:

- **Distinctive:** Algorithms should be distinctive. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Input:** If an algorithm says to take input, it should be well-defined inputs.
- **Well-Defined Output:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Practical:** The algorithm should remain simple and practical, such that it can be executed upon with available resources. It should not contain a non-existent technology.
- **Finite:** The algorithm must be finite, e.g. it should terminate after finite time.
- **Independent:** The algorithm designed should be independent, e.g. normal instructions that can be implemented in any language, and yet the output will be same.



The next question is, how do we design algorithms? In order to write an algorithm, the following factors are required:

- A clear definition of the problem that is to be solved by the algorithm.
- The constraints of the problem that must be considered while solving the problem.
- The input to be taken to solve the problem.
- The output to be expected when the problem is solved.
- The solution to the problem, within the given constraints.

Methods to help formulate algorithms:

- pseudocode - helps "think" out a problem or algorithm before trying to code it
- flowcharting - graphical way to formulate an algorithm or a program's flow
- stepwise refinement (top-down design) of algorithms

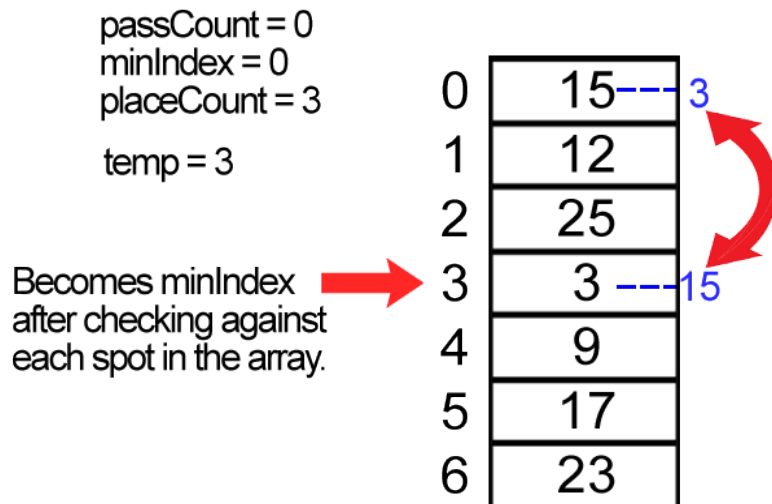
At this point, I've covered several important concepts regarding algorithms and the creation of lists to help us get through the process. Now we look at how to create an algorithm to perform a selection sort in an array.

- 0) Point to the first location
- 1) Make a pass through the list, from the current location, looking for the smallest number
- 2) Exchange the smallest with the current location being pointed to
- 3) Move the current location point to the next location
- 4) Repeat 1 – 3 until the next to the last location is reached.

```
void SelSort(int list[ ], int length)
{
    int temp;
    int passCount;
    int placeCount;
    int minIndex;

    for (passCount = 0; passCount < length - 1; passCount++)
    {
        if (list[placeCount] < list [minIndex]) minIndex = placeCount;
    }
    temp = list[minIndex];
    list[minIndex] = list[passCount];
    list[passCount] = temp;
}
```

The if statement of this algorithm goes through the array and finds the smallest number. Underneath the for loop we move the smallest value to the top element swapping the location with the existing value placing the smallest value to the top of the array.



Now that we have our algorithm in place, let's look at an example of how we'll implement it.

```

#include <iostream>
#include <iomanip>

using namespace std;

void SelSort(int list[ ], int length);

void main()
{
    char junk;
    int i;
    const int length = 9;
    int list[length] = {178, 3, 876, 32, 11, 765, 12, 77, 1};

```



```

    SelSort(length);

    for (i = 0; i < 9; i++)
        cout << list [i] << endl;

    cin >> junk;
}

```

Everything in this function goes before the for loop and is then put into SelSort



```

void SelSort(int list[ ], int length)
{
    int temp;
    int passCount;
    int placeCount;
    int minIndex;

    for (passCount = 0; passCount < length - 1; passCount++)
    {
        minIndex = passCount;
        for (placeCount = passCount + 1; placeCount < length; placeCount++)
        {
            if (list[placeCount] < list[minIndex]) minIndex = placeCount;
        }
        temp = list[minIndex];
        list[minIndex] = list[passCount];
        list[passCount] = temp;
    }
}

```

Classes and Objects

Classes are an expanded concept of data structures. Like data structures, they contain data members, but they can also contain functions as members. Nearly everything in C++ is associated with classes and objects, along with its attributes and methods. Attributes and methods are essentially variables and functions that belong to the class. These are typically referred to as *class members*. A class is a user-defined data type that we can use in our programs, and it will function as an *object constructor*, or a blue print for creating objects.

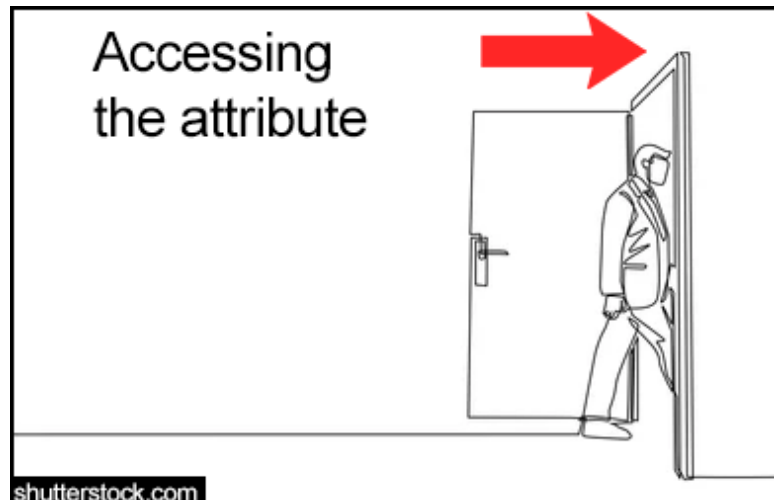
To create a class, like other data types in programming you have to declare it and in this instance you would just use the keyword *class*. Here is an example of creating a class called “MyClass”:

```
class MyClass
{
    public:
        int myNum;
        string myString;
};
```

- We start by creating a class called `MyClass`, using the keyword, `class`.
- The `public` keyword is the access specifier, which specifies that members (*attributes and methods*) of the class are accessible from outside the class.
- Inside of the class, we have an integer variable `myNum` and the string variable `myString`.
Note: Anytime variables are declared within a class, they are called **attributes**.

In C++, an object is created from a class. Since we currently have a class called `MyClass` we can use this to create objects. To create an object of `MyClass` we specify the class name, followed

by the object name. To access the class attributes of `myNum` and `myString`, we use the dot syntax, ‘.’ on the object. If you’ve observed code over time, you’ve likely seen the dot (*period symbol*) between data attributes, but maybe haven’t known what it was. Accessing objects and attributes is the instance we use this. Visualizing this is like walking through a door to access the other attribute. By placing the ‘.’ between the attributes



envison this as an execution to access the other attribute/object like walking through a door. Let’s say for example that I create an object from `MyClass` called `myObj` we would perform this as `myObj.myNum`. I haven’t spoken much about object-oriented programming (*OOP*) in this guide, but *OOP* is where classes shine the brightest. As opposed to procedural programming, where code with lots of functions and variables often runs the risk of being lengthy and redundant, *OOP* provides a clear structure for the storage and access management of variables and functions. A single class can logically group code into domain-specific areas. Object-oriented programming keeps the code cleaner, and code execution is faster as a result. The question is, why are classes relevant to *OOP*? Object-oriented programming could not exist without classes, as classes are the bread and butter of *OOP*. Thanks to classes and their ability to give you a logical way to group code and shape it around objects and data, the user or program can reuse the same lines of code within a C++ class to generate the necessary output.

The fact that C++ was built around OOP is one of the main reasons the programming language rose to prominence. Going off of my previous examples, let's look at a larger example of how we would implement objects with my attributes.

```
class MyClass
{
    public:
        int myNum;
        string myString;
};

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

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

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

This example shows how we can create multiple objects of one class by creating a car class with some attributes:

```
class Car
{
    public:
        string brand;
        string model;
        int year;
};

int main()
{
    // Create an object of Car
    Car carObj1;
    carObj1.brand = "Infiniti";
    carObj1.model = "M35";
    carObj1.year = 2006;

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

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

In C++, classes provide a great deal of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., the state without actually knowing how the class has been implemented internally. Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring a change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes to the data. If the implementation changes, only the class code needs to be examined to see what effect the change may have. **If data is public, then any function that directly accesses the data members of the old representation might be broken.** Here is an example with public and private attributes.

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

class Adder
{
public:
    // constructor
    Adder(int i = 0) {
        total = i;
    }

    // interface to outside world
    void addNum(int number) {
        total += number;
    }

    // interface to outside world
    int getTotal() {
        return total;
    };

private:
    // hidden data from outside world
    int total;
};

int main()
{
    Adder a;

    a.addNum(10);
    a.addNum(20);
    a.addNum(30);

    cout << "Total " << a.getTotal() <<endl;
    return 0;
}
```

What is Recursion?

You may have heard the joke, “To understand recursion you must first understand recursion.” It goes in line with a common definition of recursion as a function that makes a call to itself. A well-represented, real-world image of recursion is a set of those Matryoshka Russian nesting dolls that fit inside of one another. Inside the doll, is another smaller doll, inside of that is an even smaller doll, and so on. A recursive algorithm is a lot like these nesting dolls. It reproduces itself with smaller and smaller examples of itself until a solution is found. That is, until there are no more dolls. What this means is that we require a function that makes recursive calls to itself. This is the definition of **Recursion**. The formula could be represented as:

$$x^n = \underbrace{X x X x X x X x \dots x X}_{n \text{ times}}$$



In the example of, “smaller versions of itself” means that the exponent is decremented each time. With the Russian doll analogy in mind, the question is, when does the process stop in the virtual universe? The short answer to that is, when we’ve reached the *base case*. The base case is the case for which the solution can be stated as non-recursive. One example is where $N = X^1$ is X is a case in which the answer is explicitly no longer recursive and terminates. In this instance, we require a *recursive algorithm*. A **recursive algorithm** is an algorithm that expresses the solution in terms of a recursive call to itself and must terminate; that is, it must have a base case. Here is an example of a recursive `Power` function with a base case and the recursive call marked. The function is embedded in a program that reads in a number, an exponent, and then prints the result.

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

int Power( int, int );

int main()
{
    int number;
    int exponent;

    cin >> number >> exponent;
    cout << Power(number, exponent);
    return 0;
}
```

The number that is being raised to power, and the exponent is the power of the number being raised.

Non-recursive call

```
int Power( /* in */ int x,
          /* in */ int n )
```

The number that is being raised to power, and the power that the number is being raised to.

Note:

What we consider about this next section is computing X to the n power by multiplying x times the result of computing x to the $n - 1$ power. The precondition is x is assigned && $n > 0$, and the function value == x raised to the power of n .

```
{
    if (n == 1)
        return x;
    else
        return x * Power(x, n - 1);
```

← Base case

← Recursive call

Each recursive call to `Power` can be thought of as creating a completely new copy of the function, each with its own copies of the parameters x and n . The value of x remains the same for each version of `Power`, but the value of n decreases by 1 for each call until it becomes 1 and terminates.

Let's look at another example with calculating a factorial using recursive algorithms with simple variables. A factorial of a number N (written $N!$) is N multiplied by $N - 1$, $N - 2$, $N - 3$, and so on. Another way of expressing factorial is $N! = N * (N - 1)!$

This expression looks like a recursive definition. $(N - 1)!$ is a smaller instance of $N!$ – that is, it takes one less multiplication to calculate $(N - 1)!$ than it does to calculate $N!$ If we can find a base case, we can write a recursive algorithm. Fortunately, we don't have to look too far since $0!$ is defined in mathematics to be 1. Let's look at how an algorithm would be created, and then how the code would be implemented for this example.

Algorithm:

```
IF n is 0
    Return 1
ELSE
    Return n * Factorial (n - 1)
```

Coded:

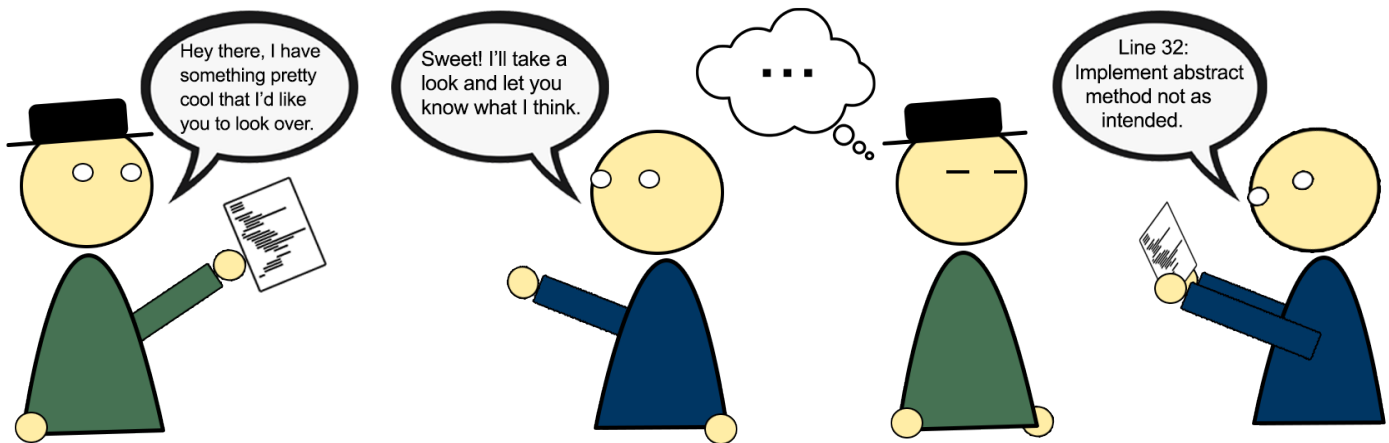
```
int Factorial ( /* in */ int n )
{
    if (n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

With these examples in mind, the question is, *why use recursion?* The short answer is, recursion can simplify more complex procedures by tracing the execution of the *values* from *variables*. By using recursion we can make programs shorter by writing less code, making our

programs syntactically easier to read and will run much faster as a result. While recursion is one of the more advanced and difficult areas to grasp in programming, it doesn't have to be. Just as continual practice in programming helps us to become more familiar with conditionals and routines, through *recursive* practice in multiple problem scenarios recursion begins to come into light. These were also extremely lightweight examples, and recursion is often used to solve much larger problems such as sorting large arrays and calculating the Towers of Hanoi.

The Developer & The Compiler

As the user (*developer*) we put in multiple requests (*coding*) to the computer (*compiler*) and if it passes the processing phase without any errors, the computer returns a successfully completed result (*functional application*).



One drawback about programming is that sometimes it feels much more convoluted than it needs to be, and in my experience, the compiler sometimes seems even more cryptic than the code. As the developer, you will inevitably have moments of discontentment when the compiler literally gives you an abstract error message that prevents your code from running. Everyone who has dabbled in code even a little knows that programming follows Murphy's Law: "*Whatever can go wrong, will go wrong.*" However, as a detailed problem-solver it is your job to locate these errors and fix them so that the compiler runs to completion and outputs your program. There are many different types of error messages depending on the languages that you're programming with. Let's take a look at a few common C++ errors that you'll likely come in contact with, what the definitions of those errors look like, specific examples of the errors, the meanings to those errors, and plausible fixes for them.

Undefined Reference

Definition:

An "Undefined Reference" is a predefined error that occurs when we have a reference to an object name (class, function, variable, etc.) in our program and the linker cannot find its definition when it tries to search for it in all the linked objects' files and libraries.

Thus, when the linker cannot find the definition of a linked object, it issues an "undefined reference" error. As clear from the definition, this error occurs in the later stages of the linking process. There are various reasons that cause an "undefined reference" error.

Example:

```
/tmp/cj142kRq.o: In function `main':  
/tmp/cj142kRq.o(.text+0x27): undefined reference to `Print(int)'  
collect2: ld returned 1 exit status
```

Meaning:

Your code called the function Print, but the linker could not find the code for it in any .o file.

Plausible Fixes:

1. You forgot to link the .o file that contains the function.
2. You misspelled the name of the function.
3. You spelled the name of the function correctly, but the parameter list is different in some way.

Undeclared Identifier

Definition:

An Undeclared identifier error is thrown by the compiler to indicate that it can't find a declaration for some identifier.

Example:

```
cal.cpp: In function `int main()':  
cal.cpp:17: `CalendarYear' undeclared (first use this function)  
cal.cpp:17: (Each undeclared identifier is reported only once for each  
function it appears in.)  
cal.cpp:17: parse error before `;' token
```

Meaning:

The compiler has not seen a definition for "CalendarYear" so it doesn't know what it is.

Plausible Fixes:

1. Ensure that you've declared the variable before the function with its proper data type.
2. Include the header file that defines the class/struct/function/etc.
3. You misspelled the name of the identifier and need to confirm the spelling.

Non-aggregate type

Definition:

Classes and structs are generically called "aggregate" types. If you get an error indicating that your class is a "non-aggregate type", then the compiler has not seen your class definition and doesn't recognize your class as such.

Example:

```
drawShape.cpp: In function `int main(int, char**)':  
drawShape.cpp:62: request for member `drawShape' in `theRect1()', which is of  
non-aggregate type `Rectangle () ()'
```

Meaning:

It's declaring a function called theRect1 returning an object of type Rectangle.

Plausible Fixes:

1. Removing the parens will nullify the function call eliminating the request to return an object.
2. Dereference the parameters as pointers, then use the attribute operator '.' or use the '->' notation.

Conclusion

I hope this guide has been a helpful guide into learning about programming, how it works, and how to see it. As I mentioned in the introduction of this guide, I have not covered anywhere close to the vast depths of what is to be learned in the field of computer science. However, I firmly believe the topics that I have covered is enough to help provide a helpful and visual perspective into understanding how to program. From here, it's up to you if you want to continue the journey. As I also mentioned in my introduction, programming is hard, and while more experience and small projects will help you to become a better programmer, the deeper you go, the more complex it becomes. This is important to realize early on. I've witnessed several programmers that can perform essential programming with quality results, but those same people eventually find themselves becoming overwhelmed on larger projects, myself included. I know first-hand what it's like to spend all night sitting at the console and trying to solve problems in programs. Eventually it does happen, but that's okay! Any last advice that I can think to provide is, make mistakes, get ALL of the errors, and learn from them, and learn how to debug them. You can't learn from your mistakes without making them. Learning about debugging is important, and you'll most likely be doing a lot of it if you jump deeper into programming. Take your time, take breaks away from the computer, and be patient. Start with the basics that I demonstrated in this guide, or go even lighter weight! Before I learned C++, I taught myself HTML. Coding with basic markup doesn't even involve programming, it's just text placement inside of containers, becoming familiar with data types, and structuring said data types. Even simple markup concepts can help you in the beginning when trying to understand how to manipulate text and other data using tags, and separating data types.

Unless otherwise specified in the images, all diagrams, code examples, and other images in this guide I created and converted digitally from my original notes and source code. If you have any questions about this guide or any other general inquiries, you can email me at

technologicguy@gmail.com

Resources Used:

- Dale, Nell – Weems, Chip. *Programming and Problem Solving With C++ (4th Edition)* – 2004
- Cplusplus.com