# From “Hello, World!” to hello.world()

## C++ Programming from Fundamentals to Objects

Matthew Donnelly

# From “Hello, World!” to hello.world()
# C++ Programming From Fundamentals to Objects

# Table of Contents

# Chapter 1: Introduction to Programming and C++

Chapter Layout



## Section 1.1: Why Programming?

At a fundamental level, learning a new subject is exactly the same as learning a new language. Subjects such as mathematics, chemistry, and computer science are all simply different languages used to solve different problems. When viewed as languages, each subject can be reduced into parts which they share in common. This involves general ideas which are needed in all fields, such as the ability to create new knowledge from previously known information and understanding how to apply knowledge to solve a problem. Learning these subjects, then, involves learning the basics of each language and understanding how to combine the basic parts into more complex and elaborate systems. The goal in doing so is to construct and express our own ideas within these fields to the greatest extent possible.

Learning how to program involves learning a language, one that is foreign and different from our own experience with human languages. Expressing a task for a computer to perform in a programming language requires not only the ability to express ideas in code, but also the ability to precisely and unambiguously define the steps to be followed to reach the end goal. Logical reasoning, design, and elegance come together in a program to create solutions for challenging problems. The more effort and time you dedicate to your study of programming, the greater the benefits you will reap. Even if programming is not your career goal, the act of developing, writing, and debugging code is a highly rewarding and challenging activity that will stretch your ability to critically think and allow you a better understanding of other subjects.

## Section 1.2: From Code to Binary

At the most basic level, computers only operate with binary data. This format is very difficult for humans to use since even the simplest of operations requires many instructions organized in a certain way. It is tedious to operate at such a primitive level, and so programming languages were created to increase the productivity of programmers.

Programming languages act as an intermediate between the idea of a program in your head and the actual program running on a computer. The process of programming is expressing

an idea in a programming language which can then be used to generate a program for a computer. By taking a **source code file** – a file containing the program you write – the compiler can generate the final **executable**, which is a binary file that the computer can use.

The goal of a programming language is to bridge the gap between the **high-level** human thought and the **low-level** operations of a processor. In order to accomplish this, programming languages are generally strict about the syntax of a program. The **syntax rules** of a programming language are the rules which define how the source code is interpreted by the compiler. Due to the preciseness of syntax rules, a single character added or removed in the source code can easily change the ability of the program to compile and produce the correct output.

If a program does not correctly follow the syntax rules of the language, the compiler will not be able to generate an executable. This type of error is called a **syntax error**, and when this occurs the compiler will display an error message which can be used to find and correct the error. Another common type of error is a **logic error**, which occurs when the program follows the syntax rules of the language, but does not perform the intended action. A logic error occurs, for example, when a program processing orders for an online store incorrectly charges $10 for a $100 item. Both types of errors represent a disconnect between the programmer and the computer.

## Section 1.3: A Basic Program

As you progress through this book, several code examples will be provided to demonstrate the concepts being covered. For this book, the online website https://repl.it/languages/cpp will be used. By using this website, you can execute the examples shown in this book without needing to install any software to your computer. In addition, you may setup an account on this site in order to save the code you are developing and continue working at a later time. Once you have the examples running and understand how they work, you should experiment with the code and observe what happens. Programming is a hands-on activity, and playing with the code on your own is the best way to learn.

```
1   #include <iostream>
2   using namespace std;
3
4   /*
5     A comment
6     spanning multiple lines
7   */
8   int main(){
9
10    //This prints "Hello, World!" to the screen
11    cout << "Hello, World!";
12
13    return 0;
14  }
```

Figure 1.3.1

As a first example, the code in figure 1.3.1 demonstrates the basic components of a C++ program. Enter the code into https://repl.it/languages/cpp exactly as it is shown. The line numbers to the left are for reference only and **should not be entered**. They

will be used to reference specific parts of the code and are included for all code examples in this book.

In order to run this code, click on the green "run" button at the top of the page. The output of the program will be on the right hand side of the screen. For this program, the output will be "Hello, World!" as shown in figure 1.3.2.

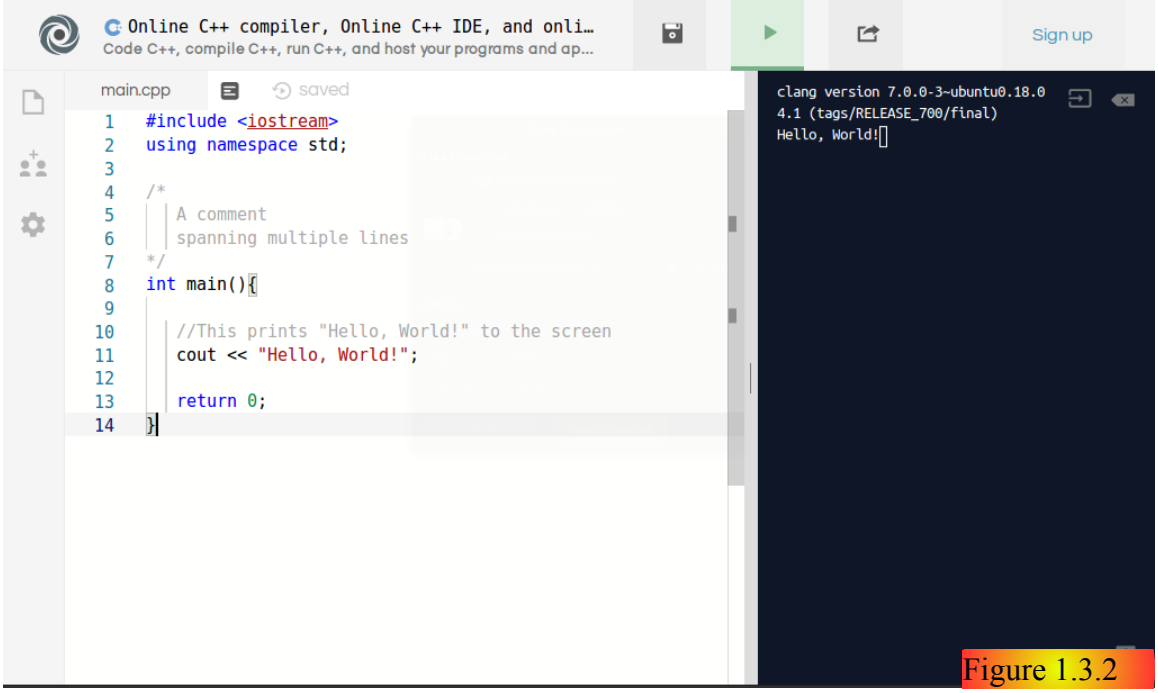


Figure 1.3.2

## Section 1.4: Components of a C++ Program

Every C++ program will have similar components in common. The job of a programmer is to combine these components into the overall program.

- Libraries

The very first line of the program in figure 1.3.1 is "#include <iostream>". The leading "#include" indicates that a library is being used in the program. The library name, **iostream**, follows in #include and is enclosed by angle brackets. A library is similar to a recipe book. By using a book of recipes that someone else wrote, we are able to use the work of others to reduce the amount of work and experimentation needed to get the end result. The **iostream** library, for example, contains code which allows the program to display text on the screen using **cout** without us programming this functionality. Libraries exist for a broad range of applications such as cryptography, networking, and graphics.

- Comments

Line 10 of figure 1.3.1 contains a comment, which is a note left by a programmer to explain what the code is meant to do. A **single line comment** begins with two forward slashes ("//") and extends to the end of the line. To span multiple lines, a **multi-line comment** can be used by beginning with "/*" and ending with "*/". All text in between is part of the multi-line comment. When debugging, it is useful to comment out code in order to temporarily remove it for the final program.

As your programs become more complex, it is good practice to place clearly written comments in order to explain what operation the code should be performing. While comments will be ignored by the compiler and will not change how the program runs, their purpose is to help you as a programmer when changing or maintaining code. Working with code written by others is one of the most difficult aspects of software development, and using comments along with proper formatting mitigates this problem. In a professional setting, there will be guidelines in place that dictate how code is formatted and commented. To become a valuable employee, you should form a habit of properly formatting and commenting your code even if commenting seems useless or the purpose of the code is obvious. As you enter the examples in this book, pay careful attention to how the code is formatted and commented. The comments in the code examples are designed to explain what is happening along side the actual code.

- Main Function

Taken as a whole, lines 8 through 14 of figure 1.3.1 form the **main function**. The main function is required for every C++ program since it is where the program will begin executing code. When all code within the main function has been executed and the return statement on line 13 has been reached, the program ends. The curly braces on lines 8 and 14 form a **code block**, and contains lines of code in between. Because this code block is being used for the main function, it is called the **body of main**.

- Formatting

Good code is well formatted code. Following a formatting standard greatly improves the quality of your code and the ease at which it can be modified and maintained by other programmers. Do not underestimate the importance of formatting when developing software in a professional setting. As you type the examples in this book, pay attention to how the code is formatted and enter it exactly as shown. Notice how the code within the body of main is indented several spaces so that it forms a visual block of code. Indentation visually separates the inner code from the outer function which allows programmers to easily see how the code is structured.

In addition to using indentation, blank lines are useful for separating pieces of code that perform different tasks. The blank lines on lines 3, 9, and 12, are used to separate different pieces of code within the source code. Just as paragraph breaks help the reader understand a book, blank lines improve readability by visually separating blocks of code.

- Program Output

Line 11 of figure 1.3.1 is responsible for printing “Hello, World!” to the screen. This is done by using **cout**, which is short for console output. By sending the text “Hello, World!” to **cout** using the stream insertion operator (“<<”), the text is displayed to the screen. Notice how the arrows are pointing from the text “Hello, World!” and pointing to **cout** as if the text were being directed to the screen. In addition, the line ends with a semicolon (“;”). Ending a line of code with a semicolon is part of the syntax of the C++ language and is used to separate the lines of code within a code block.

# Chapter 2: Variables and Types

Chapter Layout



## Section 2.1: Why are Variables Needed?

The goal of developing software is to process data. In order to do this, the memory inside of a computer is used to store data as it is being manipulated by the program. However, directly using memory is difficult since the address of the data in memory needs to be handled and managed. Programming languages remove this difficulty by allowing programmers to use a far easier solution: **variables**. Instead of needing to understand how memory is used and addressed, variables allow programmers to use names to reference data.

This is similar to how geographical locations can be represented. Consider the difference between specifying the location of your house in terms of GPS coordinates verses referring to it simply as “My house”. When using GPS coordinates, each number must be understood and correctly interpreted in order to be useful. While precise, GPS coordinates are not useful for practical purposes. Instead, names such as “My house” are assigned to locations and used to reference them at a later time. Referencing locations by name requires far less knowledge in order to interpret correctly, which means it is easier to use. In exactly the same way, variables ease the burden on programmers since they are referred to by name instead of by address.

## Section 2.2: What are Types?

Internally, data is still accessed using memory addresses, but this process happens automatically behind the scene. Because programmers do not need to manage the underlying memory, variables can be thought of as boxes that have names and can hold pieces of data.

If variables are boxes that contain data, then the purpose of a box is its **type**. Just as boxes are used for certain purposes – from shipping boxes to moving boxes to tool boxes – variables in C++ have a certain purpose depending on their type. One of the most common types of data handled by C++ programs are integers. In order to manipulate integers, the type **int** can be used. Variables of type **int** have a specific purpose, which is to hold integer numbers, and cannot be used to hold other types of data such as decimal numbers or text. In order to hold these other forms of data, different types are required. Holding decimal values, for example, requires the type **double**.

Besides numbers, text is an important type of information for a program to handle. To hold a single character, such as 'a' or 'Z', the data type **char** can be used. By grouping together variables of type **char**, words and sentences such as "Hello, World!" may be formed. By grouping together characters, an aggregate type called a **string** is created. Strings are commonly used in programs to interact with the user and to store information, and their use will be covered in a future chapter.

Variables of type **int**, **double**, and **char** are called **primitive types** since they are the most basic types of variables that the C++ language provides. Primitive types may be though of as basic construction materials such as concrete, glass, and wood. While simple, they form the basic building blocks for more sophisticated and elaborate programs. By combining primitive types, it is possible for programmers to create new types of variables that have specific purposes within a program. The chapter on objects will show how this may be done. For now, primitive types will primarily be used in example programs. Fundamentally, the primitive types just hold numbers, as the table below shows. Each type has a given name, the size that it occupies in memory, and the range of numbers that it may hold.

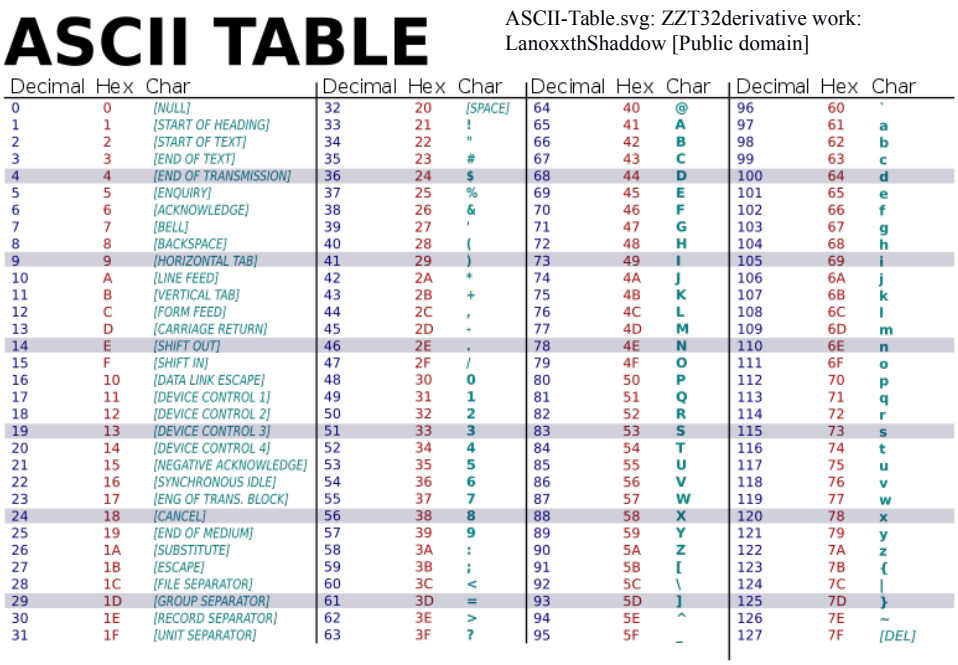
Figure 2.2.1

| Type Name | Type Size | Range of Values |
|---|---|---|
| **char** | 1 byte | 0 to 255 (see below) |
| **int** | 4 bytes | -2147483648 to 2147483647 |
| **unsigned int** | 4 bytes | 0 to 4294967295 |
| **float** | 4 bytes | +/- 3.4 e +/38 |
| **double** | 8 bytes | +/- 1.7 e +/ 308 |

The range of numbers that may be held by a certain type is based on the amount of memory required for that type. Since the amount of memory within a computer is finite, the range of values that can be stored is finite as well. An integer, for example, may not contain a value larger than 2147483647 or less than -2147483648.

While variables of type **char** hold numbers, the numbers are interpreted based on their **ASCII** values. ASCII characters are numerical representations that relate numbers to letters used in the English language. The ASCII table below shows the relationship between a number stored in a **char** type and the character that it represents.

Figure 2.2.2



ASCII TABLE

ASCII-Table.svg: ZZT32derivative work: LanoxxthShaddow [Public domain]

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

By using this table, the number 67 listed under the "Dec" column can be interpreted as the character 'C'. For a string of characters, such as "Computer", the ASCII values would be 67 for 'C', 111 for 'o', 109 for 'm', 112 for 'p', 117 for 'u', 116 for 't', 101 for 'e', and 114 for 'r'.

The type of a variable defines its purpose within a program. Learning to use C++ involves not only learning the syntax of the language, but also in understanding how to choose the right type for data. When handling numbers, for example, consider if integers or decimals will be handled. The distinction will determine if an **int** or a **double** type should be used.

## Section 2.3: Declaration and Assignment

In order to use a variable, it must first be **declared.** Declaring a variable involves specifying the type of the variable as well as the name it will be given. Once a variable has been declared, it can be **assigned** a value using the assignment operator ("="). The program in figure 2.3.1 demonstrates the process of declaring and assigning to a variable called **myInteger**. Enter

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Create a variable named "myInteger" of type int
6    int myInteger;
7
8    //Assign the value 10 to the variable
9    myInteger = 10;
10
11   //Print out the content of myInteger
12   cout << "myInteger = " << myInteger;
13
14   return 0;
15 }
```

Figure 2.3.1

this code exactly as shown into https://repl.it/languages/cpp and click the green "run" button to see the result.

Line 6 of figure 2.3.1 declares the variable. The line starts with the type **int**, which specifies that the variable being declared will be of type integer. The variable name **myInteger** follows the variable type and is used to reference this variable throughout the program.

Line 9 assigns the integer value 10 to the variable **myInteger** using the assignment operator. Because variables can be thought of as boxes with names, assigning to a variable is similar to placing an item into a box. This line, then, can be thought of as storing the value 10 into the box called **myInteger** where it will be held as the program runs.

Line 12 of figure 2.3.1 uses **cout** and the stream insertion operator ("<<") to send both the text "myInteger = " and the value of **myInteger** to the screen. By separating the items sent to **cout** using the stream insertion operator, multiple items may be sent to the screen at the same time. The program in figure 2.3.2 demonstrates this by sending three words to the screen in a single line by separating the output with the stream insertion operator.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    //The stream insertion operator may be used to
6    //separate multiple things being displayed.
7    cout << "first" << "second" << "third" << endl;
8
9   return 0;
10 }
```

Figure 2.3.2

In addition, the variable **myInteger** in figure 2.3.1 is not surrounded by double quotes when sent to **cout**. An unquoted variable name will be replaced with the value that the variable contains. If the variable name were in quotes, such as "myInteger", then the text "myInteger" would be displayed to the screen. Since the goal of that line is to display the value stored in **myInteger**, the variable is left unquoted.

A variable can be assigned to multiple times throughout the program. Each time an assignment occurs, the old value stored is overwritten by the new value. The program in figure 2.3.3 highlights this by assigning the value 10 to **myInteger** and then assigning the value 99.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Declare a variable called myInteger of type integer
6    int myInteger;
7
8    //Assign a value of 10 to the variable
9    myInteger = 10;
10
```

Figure 2.3.3

```
11    cout << "myInteger = " << myInteger << endl;
12
13    //Now assign the value 99 to myInteger
14    myInteger = 99;
15
16    //Print out the variable again.  Since the value was
17    //changed to 99 in the line above, this will display 99.
18    cout << "myInteger is now " << myInteger;
19
20    return 0;
21 }
```

Figure 2.3.3 (cont.)

The variable is declared on line 6, and then assigned the value of 10 on line 9, just like the last example. On line 14, however, the variable **myInteger** is re-assigned the value 99. By assigning this new value, the old value of 10 is overwritten and cannot be retrieved. As long as a variable is not reassigned, it will continue to hold the same value when the program runs.

In addition, the code in figure 2.3.3 uses **endl** when sending data to **cout**. The name **endl** stands for end line, and results in the cursor being moved to the next line on the screen when sent to **cout**. This is similar to what happens when the enter key is pressed in a word processor. Once this program has been entered into the online interface and is working correctly, remove "<< endl" from line 11 and observe how the output changes.

When a value is directly entered into a program, such as the numbers 10, 99, and "myInteger is now" in this code example, it is called a **literal**. Literals, like variables, have an associated type which governs how they may be used within a program. The numbers 10 and 99 in figure 2.3.3, for example, are examples of integer literals. As integer literals, the values may be assigned to variables of type **int**. The text "myInteger is now" is a string literal, and could not be assigned to a variable of type **int** due to the mismatch in types.

## Section 2.4: Initialized vs Uninitialized Variables

The code from figure 2.3.3 declared and assigned values to the variable called **myInteger**. If the variable were never assigned to, there is no guarantee what value it will contain. A variable that has not been assigned a value could be anything, and is called an **uninitialized variable**. If used in other parts of the program, an uninitialized variable could potentially cause logic errors that would be difficult to find. When writing thousands of lines of code, forgetting to initialize a variable is easy to do and may cause problems in other parts of the program. Therefore, it is wise to both declare and assign to a variable at the same time. This can be accomplished by

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     //This both declares and assigns a
6     //value to the variable myInteger
7     int myInteger = 10;
8
9     //Display the variable's value
10    cout << myInteger;
11
12    return 0;
13  }
```

Figure 2.4.1

combining both declaration and assignment in the same line. The code in figure 2.4.1 performs **declaration with assignment** with the variable **myInteger**. In this example, the variable is both declared and assigned a value on the same line to prevent the variable from being uninitialized.

## Section 2.5: Naming Variables

The C++ programming language limits what a variable may be named. In order for the compiler to correctly read a program, there are keywords and symbols which may not be used as variable names in your program. For example, a variable could not be named "<<" since this has already been defined as the stream insertion operator. Additionally, a variable could not be named "int" since this is a keyword defined in the language. The first restriction on variable names is that they cannot be keywords already recognized by the language. As you learn more about the C++ language, there will be various keywords that are used for control structures such as if statements, while loops, and for loops. These all use keywords that are reserved by the language itself. The second restriction on variable names is that variables cannot contain special characters such as "#" or "$". This includes spaces as well. For all variables declared in C++, the rule on naming a variable is that the name may only be composed of upper and lower case letters, the underscore ("_"), and numbers as long as the variable name does not begin with a number.

## Section 2.6: Using Variables

After a variable has been declared and assigned a value, it may be used to hold data being processed by the program. Operations can then be performed on the variable to manipulate the data. Some common operations on variable of type **int** are addition, subtraction, division, multiplication, and assignment. Variables may also be used in other applications that are covered in future sections, such as conditional expressions and storing input from a user. The program in figure 2.6.1 demonstrates multiple

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare and initialize myVariable
6     int myVariable = 5;
7
8     //myVariable is assigned the value of itself plus 1
9     myVariable = myVariable + 1;
10    cout << "myVariable is " << myVariable << endl;
11
12    //The second way of adding 1
13    myVariable += 1;
14    cout << "myVariable is now " << myVariable << endl;
15
16    //A third way to add 1 using the increment operator ("++")
17    myVariable++;
18    cout << "Finally, myVariable is " << myVariable << endl;
19
20    return 0;
21  }
```

Figure 2.6.1

ways of operating on a variable to increment its value by 1. While a simple example, this highlights the fact that there are often multiple correct ways to accomplish the same thing in a programming language. Part of becoming a professional programmer is learning when to use one technique over another in specific instances.

On line 9 of figure 2.6.1, the variable named **myVariable** is incremented by assigning the value of itself plus 1. When evaluated, the value of **myVariable** on the right hand side of the assignment operator evaluates to 5. By replacing the variable name with its value, the line becomes "myVariable = 5 + 1", which simplifies to "myVariable = 6". The assignment is then performed, and **myVariable** will have a value one larger than before.

Line 13 demonstrates an alternative way by using the plus equals operator ("+="). The plus equals operator is a shorthand notation used when adding and assigning a value to a variable at the same time. Similar shorthand operators exist for subtraction with assignment ("-="), multiplication with assignment ("*="), and division with assignment ("/=").

Adding 1 to a variable is a common operation in C++, and a specific shorthand operation has been developed. The plus-plus operator ("++") on line 17 is equivalent to the other two methods of increasing the variable by 1. The value may also be decreased by 1 using the minus minus operator ("--"). All three forms perform the same operation, but programmers often prefer the short hand notations when writing large volumes of code. The following table shows these shorthand operators and the equivalent expression.

Figure 2.6.2

| **Operator** | **Equivalent Expression** |
|---|---|
| variable += 2 | variable = variable + 2 |
| variable -= 2 | variable = variable - 2 |
| variable *= 2 | variable = variable * 2 |
| variable /= 2 | variable = variable / 2 |
| variable++ | variable = variable + 1 |
| variable-- | variable = variable - 1 |

In addition, variables may be added, subtracted, multiplied, and divided together and the result may be assigned to other variables.

Figure 2.6.3

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //This is multiple declaration.  Multiple variables are being
6    //declared and assigned to on the same line.
7    int number1 = 1, number2 = 2, number3 = 0;
8    number3 = number1 + number2;
9    cout << "number1 + number2 = " << number3 << endl;
```

```
10    //Notice that the values in number1 and number2 did not change.
11    //When "number1 + number2" is evaluated, it results in
12   //a value and does not assign to either variable.
13   cout << "number1 = " << number1 << " number2 = " << number2 << endl;
14
15    //Subtraction using variables.  Result is stored in number3
16    number3 = number1 - number2;
17    cout << "number1 - number2 = " << number3 << endl;
18
19    //The result of an operation can be stored into a variable.
20    //Here, the result of number1 multiplied by number2 is assigned to number3
21    number3 = number1 * number2;
22
23    //DANGER! DANGER! DANGER!
24    //This is integer division and does not behave as you would expect.
25    number3 = number1 / number2;
26    cout << "1 / 2 is " << number3 << endl;
27    cout << "1 / 2 is " << 1 / 2 << endl;
28    return 0;
29  }
```

Figure 2.6.3 (cont.)

Line 7 in figure 2.6.3 uses **multiple declaration**, which is another shorthand notation that is used to declare and initialize multiple variables of the same type. The two pieces of code in figure 2.6.4 declare and initialize the three variables in exactly the same way.

```
int number1 = 1, number2 = 2, number3 = 0;
```

Figure 2.6.4

```
int number1 = 1;
int number2 = 2;
int number3 = 0;
```

To understand how each operation is evaluated in figure 2.6.3, replace the variable names on the right hand side of the assignment operator with their values. The result on the right hand side is then assigned to the variable on the left hand side. For example, line 8 contains the expression "number 3 = number1 + number2". Since the value stored in **number1** is 1 and the value stored in **number2** is 2, the names can be replaced with the values. As a result, the expression resolves to "number3 = 1 + 2". The addition is then calculated, and the resulting value of 3 is assigned to the variable **number3**.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int number1 = 1, number2 = 2;
6     double result = number1 / number2;
7
8     cout << "result is " << result;
9
10   return 0;
11  }
```

Figure 2.6.5

The division performed on lines 25 and 27 of figure 2.6.3 does not result in the expected value. The result will be 0 instead of .5 since **integer division** is being performed. The reason why this

happens can be understood by referring to the types of the numbers being divided. The numbers 1 and 2 stored in number1 and number2, respectively, have the type of **int**. The result of the operation will have the same type of **int** as well. But, since the result of dividing 1 by 2 is a decimal value, the decimal portion is **truncated** in order to be an integer. Truncation occurs when the decimal portion of the number is removed, which results in the final value being different from what we would expect. Truncation means the result of dividing 9999 by 10000 will also be 0, since the result as a decimal will be 0.9999, which is truncated to 0 in order to be of type **int**.

To obtain the correct result for this division, the result would need to be stored in a variable that can handle decimal values. Variables of type **double** are able to do this, and the program in figure 2.6.5 attempts to solve the division problem by storing the result in **result**. However, even when assigned to a variable that can hold decimal values, the result is still zero. The issue is that **number1** and **number2** are of type **int**, and the result will be of the same type. To fix this problem, the type for each of these two operands would need to be changed from **int** to **double**.

## Section 2.7: Changing Types with Casting

To fix the previous problem involving division, the variables involved may be **casted** to be of type **double**. Casting the value of **number1** and **number2** to **double** causes the division to occur between values of type **double.** Since decimals may be held in variables of this type, the result will be .5.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main(){
6     int number1 = 1, number2 = 2;
7     double myResult = (double)number1 / (double)number2;
8
9     cout << myResult;
10
11    return 0;
12  }
```

Figure 2.7.1

To perform a cast, the desired type is enclosed by parenthesis and placed in front of the variable being casted. The code shown in figure 2.7.1, for example, casts the value of **number1** and **number2** into type **double** and then uses the result in the division. Casting does not change the type of variable; the variables **number1** and **number2** are still of type **int**. Casting interprets the integer values as decimal values, and only for the expression that the cast is used.

An integer can be converted to a decimal without causing a loss in precision. On the other hand, a decimal value could be truncated if casted to an integer. The program in figure 2.7.2 illustrates this by assigning the value stored in **myDouble** to the integer **myInteger**. Because the decimal number stored in **myDouble** is converted into an integer, the number 1.9 is truncated to 1 and stored in **myInteger**. As a result, the number in **myDouble** and **myInteger** are not equal.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare and assign 1.9 to a variable of type
6     //double called myDouble
7     double myDouble = 1.9;
8
9     //Cast the value of myDouble (which is 1.9) into an
10   //integer (which would be 1) and then assign to myInteger
11    int myInteger = (int)myDouble;
12
13    cout << myInteger;
14
15    return 0;
16  }
```

Figure 2.7.2

## Section 2.8: Input with Variables

By using **cout**, text and variables could be displayed to the screen. To retrieve data from the user, **cin** – short for console input – may be used. Reading a value from the user and storing the value into a variable allows a program to perform actions based on user input. The program in figure 2.8.1 shows how this can be implemented by prompting the user for an integer and then storing the value to the variable **myInteger**.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare an integer called myInteger
6     int myInteger;
7
8     //Prompt the user for a value
9     cout << "Enter an integer: ";
10
11    //Read in the value the user typed
12    cin >> myInteger;
13
14    //Display the user's input
15    cout << "You entered " << myInteger;
16
17    return 0;
18  }
```

Figure 2.8.1

When line 12 is reached, the program will pause until the user enters a value and presses the enter key. The stream extraction operator then takes the value from **cin** and stores it into the variable **myInteger**. In contrast to using the stream insertion operator ("<<") for outputting data, the stream extraction operator (">>") is used to pull data from **cin** and store the result into a variable. To distinguish between the two operators, notice that the arrows point in the direction of where the data is moving.

If the user enters the incorrect information, such as entering text when the program is asking for a number, the program will not behave correctly. In the future, input validation will be used to ensure that the input is valid and re-prompt the user if needed.

## Section 2.9: Constants

Variables are extremely useful since they can be used to label and hold data as it is processed by the program. **Constants**, on the other hand, are assigned a value and cannot be reassigned. While constants may seem useless with this limitation, they are important for storing data that should not change as the program runs. For example, instead of the value of **pi** being hard-coded as 3.14 into a program, a constant could be used instead. This provides a name to an otherwise meaningless number and helps other programmers to understand what a program is performing.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare a constant variable of type double
6     //and assign the value 3.14 to it.
7     const double pi = 3.14;
8     double diameter = 2;
9
10    cout << pi * diameter;
11
12    return 0;
13  }
```

Figure 2.9.1

Constants are declared just like a normal variable, but with the keyword **const** placed before the declaration. If a constant is reassigned within the program, an error will occur and the program will not compile. Constants should be used when a value should not change as the program runs, such as **pi** in figure 2.9.1. Instead of relying upon ourselves or other programmers to not assign a value to **pi**, a constant locks in the value and prevents it from being altered.

By providing names to otherwise meaningless numbers, constants eliminate **magic numbers** in a program. A magic number is a value used in a program, like 7.352, without explanation of what the value means or is used for. By assigning a name like **radius** to the value 7.352, it is easier for other programmers to recognize what the value represents.

## Section 2.10: Variable Scope

All variables declared in C++ have an associated **scope**, which defines what parts of a program may access the variable. Variable scope works in a similar way to how a name refers to a person. The name "John", for example, may refer to one person in a certain classroom, but may refer to a different person in a different classroom. In some places, no person may have the name "John". In exactly the same way, every variable has an associated scope where the variable can be referred by other code.

Variables declared within a code block have **block scope,** and the variable name can only be referenced by code contained within that block. This is called a **local variable**.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Start of code block
6     {
7       //A variable declared inside of a block
8      int insideOfLocalBlock = 10;
9
10     //Referencing this variable inside of the
11     //block works since it in the same scope
12     cout << insideOfLocalBlock;
13    }
14
15  //This will fail to compile since the
16  //variable referenced is in a different
17  //code block.  Uncomment this and
18  //see the result
19  //cout << insideOfLocalBlock;
20
21  return 0;
22  }
```

Figure 2.10.1

Alternatively, a variable can be declared outside of any code block, which means it will have a **global scope** and can be referenced by any part of the program. This is called a **global variable**. Global variables are often seen as bad practice since any part of the program may change the value of a global variable. Because of this, it is difficult to find bugs in projects that involve tens of thousands of lines of code. While global variables are bad practice, constants that have a global scope are not since their values cannot change. Figure 2.10.2 declares a global variable named **globalVariable.** This variable can be accessed by any part of the program.

```
1   #include <iostream>
2   using namespace std;
3
4   //A global variable.  These should be
5   //avoided as much as possible
6   int globalVariable = 10;
7
8   int main(){
9     //The global variable can be accessed
10    //by any part of the program
11    globalVariable = 99;
12    cout << "Global variable is " << globalVariable;
13
14    return 0;
15  }
```

Figure 2.10.2

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //First code block
6     {
7       int myVariable = 10;
8     }
9
10    //Second code block
11    {
12      //This is in a different scope, so it does not
13      //conflict with
14      //the declaration above.
15      int myVariable = 10;
16    }
17
18    return 0;
19  }
```

Figure 2.10.3

If two variables with the same name were declared within the same scope, the names would conflict and the program would not compile. While it is not possible to have variables with the same name in the same scope, two variables in different scopes are allowed to have the same name as the code in figure 2.10.3 illustrates.

Though the variable names are identical, the program compiles correctly since they occupy different scopes. If **myVariable** were declared twice in the same scope, however, the compiler would issue an error since the same variables with the same name was already declared.

Finally, code blocks may be nested, and variables declared in outer code blocks can be accessed from inner blocks. #Figure 2.10.4 declares a variable called **myVariable** within the outer code block, which is then accessed by code within the inner block. In this example, the code on line 13 can access the variable since it is contained within the outer code block where **myVariable** is declared

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Outer code block
6     {
7       int myVariable = 10;
8       //Inner code block
9       {
10        //The variable myVariable is defined in the
11        //code block that encloses this one, so
12        //it can be referenced.
13        cout << myVariable;
14      }
15    }
16    return 0;
17  }
```

Figure 2.10.4

In future chapters, code blocks will be used for various control structures. In every case, the variables being used

within the code blocks will always have an associated scope which dictates where the variable may be accessed.

## Section 2.11: Formatted Output with cout

When displaying numbers and text to the screen, the output may be controlled more precisely by using **formatting specifiers**. Formatting specifiers direct how information is presented on the screen, such as how many digits should be included in a decimal number or justifying the output within a field. The code needed to perform formatting is contained within the **iomanip** (short for io manipulation) library.

```
1   #include <iostream>
2   //Used for formatting specifiers
3   #include <iomanip>
4   using namespace std;
5
6   int main(){
7     //By placing setw(20) into the stream, the text will be
8     //aligned in a field of 20 spaces.  The specifier right is then
9     //placed into the stream so that the text is aligned to the
10    //right in the field. A field of 20 characters looks like
11    //"                  ".  By using the right specifier, the text
12    //is placed to the right "            Hello"
13    cout << setw(20) << right << "Hello";
14
15    cout << " world";
16
17    return 0;
18  }
```

Figure 2.11.1

Output can be justified within a field by using the **setw** formatting specifier along with the size of the field. When **setw** is used, the output may be aligned to the right or left in the field by using either **right** or **left**. Figure 2.11.1 demonstrates this using the **right** specifier, and figure 2.11.2 shows the same code, but using the **left** specifier.

```
1   #include <iostream>
2   //Used for formatting specifiers
3   #include <iomanip>
4   using namespace std;
5
6   int main(){
7     //By placing setw(20) into the stream, the text will be
8     //aligned in a field of 20 spaces.  The specifier right is then
9     //placed into the stream so that the text is to the right of the field.
10    //A 20 character field would look something like this
11    //"                  ".  By using the right specifier, the text is
12    //placed to the right "Hello           "
13    cout << setw(20) << left << "Hello";
14
15    cout << " world";
16
17    return 0;
18  }
```

Figure 2.11.2

By default, **setw** will pad the output with spaces. This can be changed by using the **setfill** specifier to set the character that is used for padding the output. Figure 2.11.3 uses **setfill** to pad the output with the "*" character.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main(){
6    //This is the same as the program just before, but the excess
7    //space is filled with the '*' character.  The output will
8    //look something like "hello*********"
9    cout << setw(20) << setfill('*') << left << "hello" << endl;
10
11    return 0;
12 }
```

Figure 2.11.3

Formatting specifiers may also be used to control how decimal numbers are presented on the screen by using the **setprecision** and **fixed** formatting specifiers. When used by itself, the formatting specifier **setprecision** sets the number of digits which will be displayed, using exponential notation if needed. Figure 2.11.4 uses **setprecision** to limit the number of digits displayed to three.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main(){
6    //The decimal number to be formatted on the screen
7    double myvalue = 12345.6789;
8    //This will display the first 5 digits.
9    //The text "12345" will be displayed
10   cout << setprecision(5) << myvalue << endl;
11
12   //This will display the first 3 digits.
13   //In order to do this, exponential notation is used
14   //The result will be "1.23e+04"
15   cout << setprecision(3) << myvalue << endl;
16
17   return 0;
18 }
```

Figure 2.11.4

When **setprecision** is used with the **fixed** formatting specifier, the behavior changes. Instead of setting the total number of digits displayed, **setprecision** with **fixed** sets the total number of digits after the decimal point. The code in figure 2.11.5 uses **setprecision** in exactly the same way as the last example, but modifies it with the **fixed** specifier.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main(){
6    double myvalue = 12345.6789;
7
8    //With fixed, setprecision now sets the number of
9    //digits after the decimal place.
10   //The result is now "12345.678".
11   cout << setprecision(3) << fixed << myvalue << endl;
12
13   return 0;
14 }
```

Figure 2.11.5

Finally, it is sometimes useful to display a number as a hexadecimal value. The code in figure 2.11.6 performs this operation by using the **hex** formatting specifier. Once the **hex** formatting specifier is sent to **cout**, all future numbers will be displayed in hexadecimal form. To change the formatting back to decimal, the **dec** specifier is sent.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main(){
6    int myNumber = 22;
7
8    //Print the number 22 as a hexadecimal value
9    cout << hex << myNumber << endl;
10
11   //Note that after using the hex specifier, all future numbers
12   //will be displayed in hexadecimal.  To switch back to
13   //using decimal, send the dec formatting specifier to cout.
14   cout << dec << myNumber << endl;
15
16   return 0;
17 }
```

Figure 2.11.6

# Chapter 3: Conditionals

Chapter Layout



## Section 3.1: What are Conditionals?

In order to function in everyday life, many decisions need to be made in response to the external world. The purpose of making a decision is to perform the correct response under a given circumstance. When driving a car, for example, a driver must make the decision of whether to drive through or stop at an intersection depending on the color of the traffic light. Programs, like people, need to make decisions based on various pieces of information in order to give the correct response, such as allowing a person with the correct password access to a file while denying access to everyone else.

Conditionals allow programmers to selectively execute code, such as choosing to execute one block of code under one circumstance and a different block of code under a different circumstance. Until this chapter, all of the previous programs have began execution at the first line of the main function and sequentially executed each line until the end of the main function was reached. Conditionals alter this **control flow**, which is the order that code is executed, by choosing what code should be executed. In order to understand conditionals and utilize them in a program, the concept of a boolean expression needs to be understood in order to test conditions within a program.

## Section 3.2: Boolean Expressions

Boolean expressions are expressions that are either true or false. For example, "the traffic light is green" is a boolean expression since the statement may be either true or false. In a program, a boolean expression may ask if the value of two variables are equal or not and choose which block of code to executed based on the result. To perform this comparison, the C++ language provides several **comparison operators** that are used to compare two values and return a boolean result. The table below lists a few common comparison operators between two

numbers and their equivalent meaning in English. For this table, the **a** and **b** could stand for numeric literals or variables.

| Comparison Operator | Meaning |
|---|---|
| a < b | The value of a is less than b |
| a > b | The value of a is greater than b |
| a == b | The value of a equals the value of b |
| a <= b | The value of a is less than or equal to b |
| a >= b | The value of a is greater than or equal to b |
| a != b | The value of a is not equal to b |

Figure 3.2.1

When assigning to a variable, the assignment operator is used ("="). When comparing two variables, the equality operator ("==") is used. These two look similar and are often confused when first learning how to use C++. To differentiate between the two, remember that in order to compare, there must be at least **two things** in order to perform a comparison. It is not possible to compare a single thing. With this in mind, comparison requires **two equals** ("==") while assignment requires only a single ("=").

The result of a boolean expression is always true or false. In C++, true may be represented as the number 1 while false may be represented by the number 0. The program in figure 3.2.2 demonstrates this by evaluating several boolean expressions, storing the result in variables of type **bool**, and displaying the result using **cout**. Variables of type **bool** can be used to hold a boolean value, and are another primitive type available in C++.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int a = 1;
6     int b = 2;
7
8     //Is a less than b?
9     bool aLessThanb = a < b;
10    cout << "a < b: " << aLessThanb << endl;
11
12    //Is a greater than b?
13    bool aGreaterThanb = a > b;
14    cout << "a > b: " << aGreaterThanb << endl;
15
16    //Is a equal to b?
17    bool aEqualTob = a == b;
18    cout << "a == b: " << aEqualTob << endl;
19
20    //Is a not equal to b?
21    bool aNotEqualTob = a != b;
22    cout << "a != b: " << aNotEqualTob << endl;
23
24    //Any non-zero value is true
25    //Try changing this value to any non-zero value.
26    bool nonZeroValue = 10;
28    cout << "nonZeroValue = " << nonZeroValue << endl;
29
30    return 0;
31  }
```

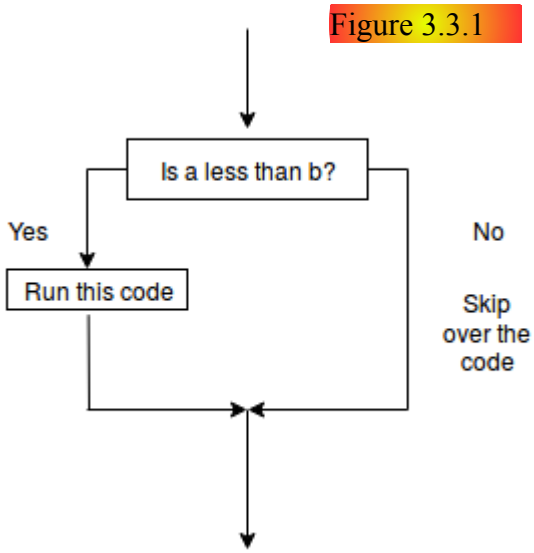
Figure 3.2.2

In C++, true is any value that does not equal 0. This means that the number 10 is true while the number 0 is false.

## Section 3.3: The if Statement

Boolean expressions are an important step towards allowing a program to make decisions based on a condition. On their own, however, boolean expressions can test a condition, but cannot act upon the result. When paired with an **if statement**, a **control structure** structure within the language, boolean expressions may be acted upon to allow **branching** within the code. Branching allows the program to choose which code is executed based on boolean expressions. The diagram in figure 3.3.1 shows the high-level overview of how branching works. The control flow begins at the top of the diagram and reaches the condition "Is a less than b?". Depending on the result, the code may be executed, or skipped over.

Figure 3.3.1




An if statement works by evaluating a boolean expression and executing the code block that follows if the result is true. If the result is false, the code block is skipped over and control flow continues after the code block. The code block that comes after the if statement is called the **body of the if statement**. When the program in figure 3.3.2 is executed, the condition "a < b" will be checked. Since the expression is the same as "1 < 2", the result is true and **the body of the if statement**, the code block that follows, is executed. Because of this, the text "a is less than b!" will be displayed. If, on the other hand, the variable **a** had the value 5 and the variable **b** had the value 1, the condition "a < b" would be the same as asking if "5 < 1", which is false. As a result, the code within the if statement would be skipped over.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int a = 1;
6     int b = 2;
7
8     //Inside is the boolean expression "a < b".  If this
9     //expression is true, then the body of the if statement
10    //will be executed.  If the expressions is false,
11    //then the code in the if statement will be skipped
12    if( a < b ){
13      cout << "a is less than b!";
14    }
15    //If "a < b" is false, execution skips over to here
16    return 0;
17  }
```

Figure 3.3.2

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int a = 2;
6     int b = 1;
7
8     //If the number a is less than the number b,
9     //then run the following code
10    if( a < b ){
11      cout << "a is less than b!" << endl;
12    }
13
14    //If the number a is greater than the number b,
15    //then execute the following code
16    if( a > b ){
17      cout << "a is greater than b!" << endl;
18    }
19    return 0;
20  }
```

Figure 3.3.3

The code may be improved further by displaying a message to indicate which variable is greater than the other. This can be accomplished by adding an additional if statement, as done in figure 3.3.3.

While adding another if statement works, there is a more efficient way to structure the code. The if statement has other forms that modify how it behaves in a program.

## Section 3.4: Chaining Conditionals

When multiple conditions must be checked, like in the last example, and only one block of code needs to be executed, the if statement can be extended into an **if..else if** statement. Adding **else if** allows multiple conditions to be tested in a row, and only the code block corresponding to the first true boolean expression is executed.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int a = 1;
6     int b = 1;
7
8     //If "a < b" is true, run the following code,
9     //then jump to the end of the if statement
10    if( a < b ){
11      cout << "a is less than b!" << endl;
12
```

Figure 3.4.1

The last code example can be re-written to use an **if..else if** statement, and an additional case added to check for equality, as shown in figure 3.4.1. The three conditions are setup on lines 10, 13, and 19. The **if...else if** statement will check the first condition on line 10, and execute the following code block if the

```
13    }else if( a > b ){
14
15      //If “a > b” is true, the line below is run
16      //and control jumps to the end
17      cout << “a is greater than b!” << endl;
18
19    }else if(a == b ){
20
21      //If “a == b” is true, then run this line
22      cout << “a equals b!” << endl;
23    }
24    //End of if statement
25
26    return 0;
27  }
```

Figure 3.4.1 (cont.)

result is true. If it is not true, then the condition on line 13 is checked and the code block is executed if true.
If the second condition is not true, then the third condition on line 19 is checked and the code is executed if the boolean expression is true. In this case, at least one of the three must be true. However, it is possible to have none of the expressions evaluate to true. When this happens, no code is executed.

This may not always be desired, and in some cases a default is needed if none of the other cases work. An example of this would be if we needed to test if a character is equal to ‘a’ or not. To do this, the code in figure 3.4.2 adds an else statement to be run by default if the first condition is not true.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     char character = 'b';
6
7     //If the character is 'a'...
8     if( character == 'a' ){
9       //...then execute this
10      cout << "Character a" << endl;
11    }else{
12      //...but if the character is not
13      //an 'a', execute this line.
14      cout << "Not character a" << endl;
15    }
16
17    return 0;
18  }
```

Figure 3.4.2

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     int grade = 55;
6
7     //If the grade is greater than 90...
8     if( grade >= 90 ){
9       //...print this out and skip all
10      //other else if and else parts.
11      cout << “You got an A”;
```

Figure 3.4.3

An if statement may contain both an **else if** and an **else** clause. The code in figure 3.4.3, for example, converts a number to a grade. This is done by comparing the grade to the borderline for the grade range, and executing the corresponding code block.

```
12
13    //If the grade is greater than 80...
14    }else if( grade >= 80 ){
15      //...print this out and skip all
16      //other else if and else parts
17      cout << "You got a B";
18
19    //If the grade is greater than 70...
20    }else if( grade >= 70 ){
21      //...print this out and skip all
22      //other else if and else parts
23      cout << "You got a C";
24
25    //If the grade is greater than 60...
26    }else if( grade >= 60 ){
27      //...print this out and skip all
28      //other else if and else parts
29      cout << "You got a D";
30
31    //If non of the other parts evaluate to true,
32    //then this is the default.
33    }else{
34      cout << "You got a F";
35    }
36
37    return 0;
38  }
```

Figure 3.4.3 (cont.)

For the grade 55, the conditions on lines 8, 14, 19, and 26 are all checked and are false. Because of this, the else part executes since it is the default when none of the other cases are true.

## Section 3.5: Nested Conditionals and Logical Operators

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int a = 1;
6     int b = 0;
7
8     //The outer if statement. This is tested first.
9     if( a > b ){
10      //The inner if statement. If the outer
11      //if statement is true, then this is tested.
12
```

Figure 3.5.1

An if statement may be placed within another if statement to create a **nested if statement**. Nested if statements are useful when multiple conditions must be true before a section of code should be executed. As an example, the code in figure 3.5.1 will only execute the contained code if both the condition "a > b" and "a != 0" are true. When this code runs, the condition on line 9 will first be checked.

```
13      if( a != 0 ){
14        //If both the outer and inner if
15        //statements are true, then this is executed
16        cout << "Yep, a > b and a != 0" << endl;
17      }
18    }
19
20    return 0;
21  }
```

Figure 3.5.1 (cont.)

If the result is true, the condition for the if statement on line 13 is checked. If the result is true again, the enclosed code is executed. If either condition is false, the code will be skipped over.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int number1, number2, number3;
6
7     //Enter first number
8     cout << "Enter the first number: ";
9     cin >> number1;
10
11    //Enter second number
12    cout << "Enter the second number: ";
13    cin >> number2;
14
15    //Enter third number
16    cout << "Enter the third number: ";
17    cin >> number3;
18
19    //If the first number is a 1 ...
20    if( number1 == 1 ){
21      //… and the second number is a 2 and ...
22      if( number2 == 2){
23        //… the third number is a 3 ...
24        if( number3 == 3 ){
25          //Then print this out
26          cout << "Correct password!" << endl;
27        }
28      }
29    }
30
31    return 0;
32  }
```

Figure 3.5.2

Conditionals can be nested arbitrarily deep. The program in figure 3.5.2 uses three if statements nested inside one another to check if all three conditions are true before displaying "Correct Password!".

The first part of this code from lines 7 through 17 read in three integers from the user and stores the result in **number1**, **number2**, and **number3**. The first if statement is encountered, and the condition "number1 == 1" is checked. If this condition is true, then the next if statement checks the condition "number2 == 2". If this condition is true, then the third if statement checks the condition "number3 == 3". If all three conditions are true, the code will run and display "Correct password!".

While it is possible to nest if statements to check multiple conditions, there is an easier approach. Instead of using three if statements to check three conditions, the three conditions may be joined together using **logical operators**.

By using logical operators, the three boolean expressions in this code can be combined into a single boolean expression that can be checked by a

single if statement. Logical operators are used to glue together many boolean expressions into a single expression. For example, the sentence "it is raining out and the traffic light is green" uses the logical operator **and** to combine the two boolean expressions "it is raining out" and "the traffic light is green". With a logical operator, the entire expression has a single value, even though it contains multiple boolean expressions. The three main logical operators in C++ are shown in the table below along with an explanation of what each means.

Figure 3.5.3

| Logical Operator | Name | Meaning |
|---|---|---|
| && | Logical and | Joins two boolean expressions and is true only if both boolean expressions are true. |
| \|\| | Logical or | Joins two boolean expressions and is true only if at least one of the expressions is true. |
| ! | Logical negation | Negates a boolean expression. When placed before a true expression, it becomes false. When placed before a false expression, it becomes true. |

Figure 3.5.4

| Logical **and** (&&) truth table | true | false |
|---|---|---|
| true | true && true = true | true && false = false |
| false | false && true = false | false && false = false |

Figure 3.5.5

| Logical **or** (\|\|) truth table | true | false |
|---|---|---|
| true | true \|\| true = true | true \|\| false = true |
| false | false \|\| true = true | false \|\| false = false |

Figure 3.5.6

| Logical **not** (!) truth table | true | false |
|---|---|---|
| | !true = false | !false = true |

The logical operators function similarly to how the English words **and**, **or**, and **not** are used in everyday language. In order to condense the three conditions in figure 3.5.2, the logical **and** (&&) operator will be used since all three conditions must be true for the code to execute. By using a logical operator, the code can be rewritten to be much shorter in figure 3.5.7.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int number1, number2, number3;
6
7     cout << “Enter the first number: “;
8     cin >> number1;
9
10    cout << “Enter the second number: “;
11    cin >> number2;
12
13    cout << “Enter the third number: “;
14    cin >> number3;
15
16    //if (number1 == 1) AND (number2 == 2) AND (number3 == 3),
17    //then the body of the if statement is executed.
18    if( ( (number1 == 1) && (number2 == 2) ) && (number3 == 3) ){
19      cout << “Correct password!” << endl;
20    }
21
22    return 0;
23  }
```

Figure 3.5.7

## Section 3.6: The switch Statement

A common pattern that occurs when using **if..else if** statements is to test if a variable equals a literal. The code in figure 3.6.1 repeatedly tests the variable **grade** for equality to the character literals ‘A’, ‘B’, ‘C’, ‘D’, and ‘F’ to display the appropriate message.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     char grade = 'A';
6
7     if( grade == 'A' ){
8       cout << "You got an A!" << endl;
9     }else if( grade == 'B' ){
10      cout << "You got a B!" << endl;
11    }else if( grade == 'C' ){
12      cout << "You got a C!" << endl;
13    }else if( grade == 'D' ){
14      cout << "You got a D!" << endl;
15    }else if( grade == 'F' ){
16      cout << "You got an F!" << endl;
17    }else{
18
```

Figure 3.6.1

```
19          cout << "Unrecognized grade!" << endl;
20      }
21
22      return 0;
23  }
```

Figure 3.6.1 (cont.)

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     char grade = 'A';
6
7     //Switch on the variable grade
8     switch(grade){
9       //If grade == 'A'
10      case 'A':
11        cout << "You got an A!" << endl;
12        //Jump to the end of the switch
13        break;
14      //If grade == 'B'
15      case 'B':
16        cout << "You got a B!" << endl;
17        //Jump to the end of the switch
18        break;
19
20      //If grade == 'C'
21      case 'C':
22        cout << "You got a C!" << endl;
23        //Jump to the end of the switch
24        break;
25
26      //If grade == 'D'
27      case 'D':
28        cout << "You got a D!" << endl;
29        //Jump to the end of the switch
30        break;
31
32      //If grade == 'F'
33      case 'F':
34        cout << "You got an F!" << endl;
35        //Jump to the end of the switch
36        break;
```

Figure 3.6.2

When code is structured like this, with **if..else if** testing whether a variable is equal to a literal several times in a row, it can be replaced by a **switch** statement.

A switch statement can only be used to compare a variable to a literal or constant using equality. A switch statement is not capable of testing if a number falls into a range.

The **break** statements at the end of every **case** statement cause the switch statement to end, and control to jump to the end. Without using **break**, the code within the switch statement would execute sequentially and multiple lines would be displayed. When the code has been entered, try commenting out the break statements and observe the result.

```
37      //This is like the else part of an if statement.
38      //Executed if none of the above matches.
39      default:
40        cout << "Unrecognized grade!" << endl;
41    }
42    //End of switch statement. This is where
43    //all of the breaks will jump to.
44
45    return 0;
46  }
```

Figure 3.6.2 (cont.)

## Section 3.7: User Input with Conditionals: Menus

By combining user input with conditionals, a menu program can be created that displays several options to a user and asks for a selection.

The code in figure 3.7.1 does just this. Lines 9 through 12 are responsible for displaying the menu to the user. Once the menu has been presented, lines 15 and 16 wait for the user to enter a selection within the menu. The program then tests which option the user entered to display the correct message.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Holds the number the user inputs
6     int userSelection;
7
8     //Display the menu for the user to see
9     cout << "Main menu" << endl;
10    cout << "   Selection 1: Display hello, world!" << endl;
11    cout << "   Selection 2: Display goodbye, world!" << endl;
12    cout << "   Selection 3: Display C++ is awesome" << endl;
13
14    //Prompt the user and get input
15    cout << "Enter selection: " << endl;
16    cin >> userSelection;
17
18    //Determine which one the user selected and execute the correct line
19    if( userSelection == 1 ){
20      cout << "hello, world!" << endl;
21    }else if( userSelection == 2 ){
22      cout << "goodbye, world!" << endl;
23    }else if( userSelection == 3 ){
24      cout << "C++ is awesome" << endl;
25    }else{
26      cout << "Invalid option" << endl;
27    }
28
29    return 0;
30  }
```

Figure 3.7.1

# Chapter 4: Iteration

Chapter Layout



## Section 4.1: What is Iteration?

A major purpose of a computer is to relieve humans of performing repetitive tasks. Until now, all of the programs presented have been run sequentially from top to bottom. The previous chapter on conditionals introduced the ability to alter the control flow of a program by selectively executing code blocks. In order to have a computer perform the same operation over and over, we need the capability to execute a block of code more than once. In C++, **loops** perform this function by continuously executing a block of code based on the result of a boolean expression. Loops alter the control flow of a program by allowing the program to jump back to the start of a code block, which allows the code to be run multiple times. Without loops, the code to perform an operation would have to be copied over several times and executed sequentially. Sequentially executing the same block of code is tedious and error-prone, in addition to being less flexible than a loop.

## Section 4.2: Looping with while

One of the main types of loops in C++ is the **while loop**. The while loop, just like the if statement, uses a boolean expression to govern how the code is executed. Just like an if statement, a while loop first tests a boolean expression and executes the following code block if the result is true. However, unlike an if expression, the while loop jumps back to the beginning once the code has been run. The boolean expression is evaluated again to decide if the code should be executed once more. If the result is true, then the code block is executed again. This continues until the boolean expression controlling the loop becomes false.

Conceptually, a while loop functions exactly like the word “while” does in English. For example, consider the sentence “while it is not raining, I will play at the park”. This sentence can be divided into three parts: the word “while”, the boolean expression “it is not raining”, and

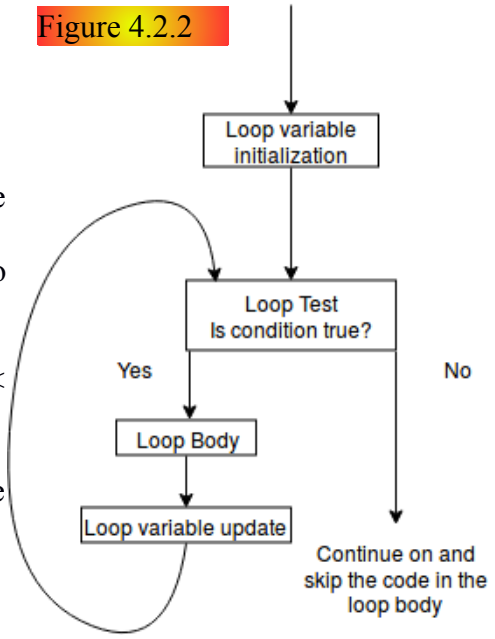
the action “I will play at the park”. So long as the condition “it is not raining” is true, the action “I will play at the park” will be performed. Once the condition “it is not raining” becomes false, the action will stop.

In exactly the same way, a while loop accepts a boolean expression and continues to execute a code block so long as the expression is true. After the loop body has been executed, the control flow of the program will jump back to the start of the while loop, check the loop condition, and run the code again if the condition is true. This process continues until the loop condition becomes false. This general process is illustrated in figure 4.2.2. Figure 4.2.1 demonstrates the use of a while loop by counting from 0 to 4.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Part 1: The loop variable initialization.
6    int loopVariable = 0;
7
8    //Part 2: The loop test using the boolean expression “loopVariable < 5”.
9    //The loop will continue as long as loopVariable is less than 5.
10   while( loopVariable < 5 ) {
11     //Every iteration will print out the value of the loop variable
12     cout << “Loop variable is “ << loopVariable << endl;
13
14     //Part 3: Loop variable update.  Add 1 to the value of the loop variable.
15     loopVariable++;
16   }
17 }
```

Figure 4.2.1

Figure 4.2.2




The three main parts of a while loop are the **loop variable initialization**, the **loop test**, and the **loop variable update**. Line 6 in figure 4.2.1 declares and initializes the loop variable with a value of 0. This variable is used in the boolean expression controlling the while loop on line 10. As long as the condition “loopVariable < 5” is true, the loop body will continue to execute. Line 15 contains the loop variable update, which is an important piece of a while loop. Since the loop only terminates when the condition “loopVariable < 5” is false, the body of the while loop must alter **loopVariable** otherwise the code will run forever. Forgetting the loop variable update is a common mistake to make, and results in an **infinite loop**. The illustration to the right shows the control flow of a while loop. The program begins at the loop variable initialization, and

then tests the loop condition. As long as the condition is true, the code within the body of the while loop will continually be executed. Once the loop test is false, the loop ends.

## Section 4.3: The for Loop

In addition to the while loop, the C++ language provides the **for loop**. Just like the while loop, the for loop contains the same three important components: the **loop variable initialization**, the **loop test**, and the **loop variable update**. The difference between a while and a for loop is where these pieces are placed. A while loop will have these three parts on different lines, with the loop variable initialization preceding the loop and the loop variable update at the end of the following code block. A for loop, on the other hand, combines all three into a single line by separating them with semicolons. The diagram in figure 4.3.2 shows the three main parts contained within the for loop.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //for(Loop variable initialization; loop test; loop variable update)
6     for(int loopVariable = 0; loopVariable < 5; loopVariable++){
7       cout << "loopVariable is " << loopVariable << endl;
8     }
9
10    return 0;
11  }
```

Figure 4.3.1

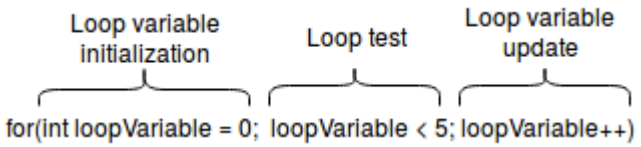



Figure 4.3.2

By convention, a for loop starts at 0 and usually counts up to, but not including, the number in the condition. The loop in figure 4.3.2, for example, starts at 0, then counts up to 4. Loops usually start at 0 instead of 1 since arrays are often processed inside of loops. In a future chapter on arrays, you will better understand the relationship between loops and array processing.

## Section 4.4: Equivalence of for and while

Functionally, while and for loops are interchangeable. Every for loop can be written as a while loop and visa versa. Since the three main parts of the loop are the same in each, one can

be converted into the other. Figure 4.4.1 shows the general layout of a while and for loop, and how to rearrange the parts to switch between the two.

```
1  <loop variable init>
2  while( <loop condition> ){
3    code
4    <loop variable update>
5  }
6
7  for(<loop variable init>; <loop condition>; <loop variable update>){
8    code
9  }
```

Figure 4.4.1

A for loop is also known as a **counting loop**, since it usually counts from 0 up to some number. The while loop, on the other hand, is known as a **sentinal loop** since it customarily uses a condition that indicates when the loop should stop. Both types of loops are interchangeable, and there is no restriction on which can be used.

## Section 4.5: Squares Using a Loop

By using loops, the squares of the numbers from 1 to 10 can be calculated. A for loop will be used since this program will be counting from 0 to 10.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    //Loop from the number 0 to 10.
6    for(int number = 0; number <= 10; number++){
7      cout << number << " * " << number;
8
9      //Multiply the number by itself
10     cout << " = " << number * number << endl;
11   }
12
13   return 0;
14 }
```

Figure 4.5.1

The program works by looping from the values 0 to 10. Inside of the loop, the loop variable is used in order to calculate the square of the numbers. After every iteration of the loop, **number** is incremented by 1. When using a for loop, the loop variable should never be assigned to in the body of the loop.

## Section 4.6: The do...while Loop

A third type of loop available is the **do..while loop**. Although this type of loop is not used as often as while and for loops, it is useful for certain applications. Both the while and for loops are examples of **pre-test loops** since they evaluate the loop condition before executing the body of the loop. The do...while loop, on the other hand, is an example of a **post-test loop** which checks the condition after the code block has been executed.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int loopVar = 5;
6
7     //This is already false, so the loop body is never executed
8     while( loopVar < 3 ){
9       cout << "This is never printed" << endl;
10      loopVar++;
11    }
12
13    return 0;
14  }
```

Figure 4.6.1

The difference between a pre-test and post-test loop is the order that the loop test and code are executed. A pre-test loop may not execute the code if the condition is false when first evaluated. The code within the while loop of figure 4.6.1 is never executed since the loop test is false to begin.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //A variable to hold the user's input
6     int userInput;
7
8     do{
9       //Prompt for and read in user's input
10      cout << "Enter number less than 5: ";
11      cin >> userInput;
12
13      //As long as the number entered is greater than or equal to 5,
14      //continue to prompt for a value.
15    }while(userInput >= 5 );
16
17    return 0;
18  }
```

Figure 4.6.2

For certain tasks, the code inside of the loop should execute at least once. Once such application is user input validation, which the code in figure 4.6.2 demonstrates.

The same code could be written with a while loop, as shown in figure 4.6.3, but the inner part of the loop would need to be duplicated twice. Once to initially ask the user for input, then again in case the number was not valid. A do..while loop guarantees that the code will be run at least once, which is why the code only needs to appear once in the body of the loop.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //A variable to hold the user's input
6    int userInput;
7
8    //Prompt for and get user input
9    cout << "Enter number less than 5: ";
10   cin >> userInput;
11
12   while( userInput >= 5 ){
13     //Prompt for and read in user's input
14     //Notice how we have to duplicate this code.
15     cout << "Enter number less than 5: ";
16     cin >> userInput;
17   }
18
19   return 0;
20 }
```

Figure 4.6.3

## Section 4.7: Nested Loops

Loops, like conditionals, can be nested. By nesting loops, a procedure that requires multiple loops can be condensed into a nested loop. As an example of how this works, consider the code in figure 4.7.1 that prints out the numbers from 0 to 4 on a single line.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Loop from 0 to 4 and print out the value of loopCounter each time
6    for(int loopCounter = 0; loopCounter < 5; loopCounter++){
7      cout << loopCounter;
8    }
9    return 0;
10 }
```

Figure 4.7.1

In order to display the line "01234" four times in a row, the loop can be copied and pasted four times in a row. Notice also with this example that the loop counters can all be called loopCounter without the names conflicting. The loop variable is local to the for loop.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Print the first time
6    for(int loopCounter = 0; loopCounter < 5; loopCounter++){
7      cout << loopCounter;
8    }
9    cout << endl;
10   //Print the second time
11   for(int loopCounter = 0; loopCounter < 5; loopCounter++){
12     cout << loopCounter;
13   }
14   cout << endl;
15   //Print the third time
16   for(int loopCounter = 0; loopCounter < 5; loopCounter++){
17     cout << loopCounter;
18   }
19   cout << endl;
20   //Print the fourth time
21   for(int loopCounter = 0; loopCounter < 5; loopCounter++){
22     cout << loopCounter;
23   }
24   cout << endl;
25   return 0;
26 }
```

Figure 4.7.2

Nested loops can be used in this circumstance to condense the four separate loops. Since the same loop is being run four times, another for loop can be used to repeat the code four times as in figure 4.7.3.

The loop on line 8 is called the inner loop since it is contained

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Repeat four times
6    for(int i = 0; i < 4; i++){
7      //Print out the line "01234"
8      for(int loopCounter = 0; loopCounter < 5; loopCounter++){
9        cout << loopCounter;
10     }
11     cout << endl;
12   }
13
14   return 0;
15 }
```

Figure 4.7.3

within another loop. The loop on line 6 is called the outer loop since it contains a loop inside. The outer loop loop variable called **i** will run from 0 to 3, which means the inner loop will be executed four times.

The code can be modified further to create a times table in figure 4.7.4. During the first iteration of the outer loop, the value of **i** will be 0. The inner loop will then iterate from 0 to 4. Each time, the outer an inner loop variables are multiplied. For the first iteration, the values 0 * 0, 0 * 1, 0 * 2, 0 * 3, and 0 * 4 are calculated. Once the inner loop ends, the outer loop starts on the next iteration where **i** equals 1. When the inner loop runs, it will loop from 0 to 4 and calculate 1 * 0, 1 * 1, 1 * 2, 1 * 3, and 1 * 4. The process repeats until the outer loop ends.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Repeat four times
6     for(int i = 0; i < 5; i++){
7       //Loop from 0 to 4
8       for(int loopCounter = 0; loopCounter < 5; loopCounter++){
9         //Multiply the inner and outer loop counters together
10        cout << loopCounter * i << " ";
11      }
12      cout << endl;
13    }
14
15    return 0;
16  }
```

Figure 4.7.4

## Section 4.8: Input Validation

When prompting the user for input, the data entered could be incorrect. A user could enter a value of 10, for example, when the program asked for a number between 0 and 5. This can be fixed by checking if the number entered by the user is within a range using an if statement. Another type of problem that can occur is if the user enters completely wrong data, such as entering the text “hello” when a number is expected. In order to recover from these errors, a loop can be used to validate the user’s input before being used in the program. When developing software in a professional setting, input validation is very important. Not only do we want our programs to withstand input from users who mistakenly enter the wrong data, but also to prevent others from deliberately crashing or breaking our programs. The program below prompts the user to enter an integer, and re-prompts if bad data is entered.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Variable to hold user input
6    int input;
7
8    //Prompt the user for an integer
9    cout << "Enter an integer: ";
10   cin >> input;
11
12   //If the user entered something wrong (like "hello"), then
13   //cin.fail() will result in true.  In the future, you will learn that
14   //cin is an object, and cin.fail() is calling the method fail() on cin.
15   //For now, just understand that cin.fail() will be true if there is a problem.
16   while(cin.fail()){
17     //Calling cin.clear() clears the error from cin
18     cin.clear();
19
20     //The ignore method will skip over the invalid input until a
21     //newline ('\n') is reached.
22     cin.ignore(80, '\n');
23
24     //We know an error occurred, so re-prompt the user
25     //If more bad data is entered, the while loop will run again
26     cout << "Error. Re-enter: ";
27     cin >> input;
28   }
29
30   cout << "You entered " << input << endl;
31
32   return 0;
33 }
```

Figure 4.8.1

The program in figure 4.8.1 uses the condition **cin.fail()** for the loop test. This statements calls a method on the **cin** object to test if reading the input has failed. On the chapter on objects, you will be introduced to the concept of an object and a method. For now, just understand that cin.fail() will be true if bad input has been entered and false otherwise. If the condition is true, the error is cleared by calling another method on cin called cin.clear(). Yet another method is called on cin on line 22 that ignores the bad input entered by the user. In particular, line 22 ignores the first 80 characters the user entered until the newline character ('\n') is reached. The newline character has exactly the same effect as **endl,** and is the result of the user pressing the enter key. Finally, the user is re-prompted on lines 26 and 27 for new input. A while loop is used instead of an if statement, because the user could enter the incorrect data again. With a loop, the condition is checked again and any new errors are processed.

# Chapter 5: Arrays and Strings

Chapter Layout



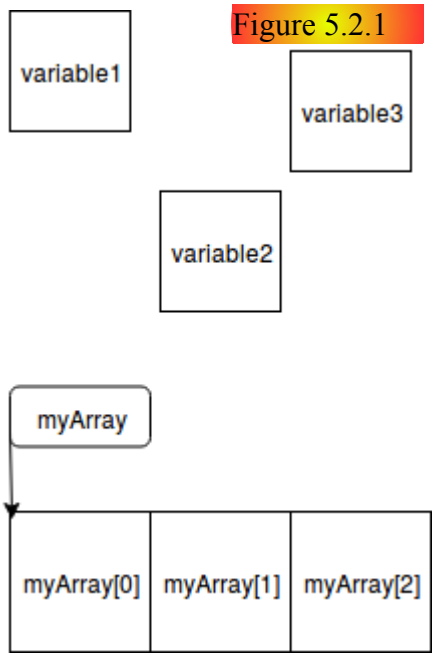
## Section 5.1: Why Arrays?

By using variables, individual pieces of data can be stored and manipulated by our program. A problem arises when the amount of data being processed becomes larger and larger. While managing a small amount of data is manageable using individual variables, processing large amounts of data becomes prohibitively complicated. If there were thousands of pieces of data to be processed, using thousands of variables would be inefficient and unmanageable.

## Section 5.2: What are Arrays?

In order to counter this problem, an **array** can be used to group together data of the **same type** and of similar purpose. For example, instead of managing many variables to hold the test scores of a student, a single array could be used. Arrays group together related data into a single unit which can then be processed by a loop. If variables are boxes that are given a name, then an array would be a line of boxes that collectively have a name and can be referenced as 1st box, 2nd box, and so on. A numerical **index** is used to specify which item in the array is being referenced. Arrays are an example of an **aggregate type**, which allows **scalar types** such as **int**, **double**, and **char**, to be grouped together.

Figure 5.2.1




An array acts by combining a number of scalar types, or even other aggregate types, into a single entity. Once combined into an array, the individual variables, called **elements**, can be accessed by providing an offset into the array.

The offset within an array is determined based on the index provided when referencing the element. Arrays used a zero-based index, where the first element within the array is at index 0. The first element is at 0 instead of 1 since the index provides an offset into the array. The first element in an array is at an offset of 0 from the beginning of the array. The second element in an array is at index 1, since the second element is one step from the beginning of the array.

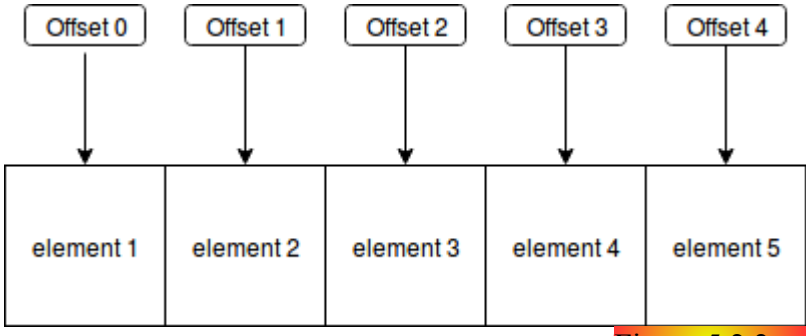


Figure 5.2.2

## Section 5.3: Declaring and Assigning to Arrays

Since an array is a collection of variables, the declaration of an array is similar to the declaration of a single variable. The only difference between declaring a single variable and declaring an array is the use of the square brackets. Figure 5.3.1 shows how an array is declared and how each element can be indexed and assigned a value.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare an array called testScores that will hold 5 integers
6     int testScores[5];
7
8     //Assign to each element in the array.
9     testScores[0] = 86;
10    testScores[1] = 95;
11    testScores[2] = 69;
12    testScores[3] = 96;
13    testScores[4] = 89;
14
15    return 0;
16  }
```
Figure 5.3.1

This code declares an array named **testScores** that holds five integers. Once the size of the array has been specified in its declaration, it cannot change as the program runs. Lines 9

through 13 reference and assigned to each element within the array by specifying the index of the element. Since the first index starts at 0, an array with a size of 5 has a maximum index of 4. For any array, the maximum valid index is one less than its size.

Although this code condenses five separate variables into an array, each individual element still needs to be initialized with a value. An array can be declared and initialized by enclosing the values for each array element by separating them with commas and enclosing them in curly braces, as shown in figure 5.3.2.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Assign testScores[0] 86, testScores[1] 95, …
6     //The value "{86, 95, 69, 96, 89}" is an array
7     //literal.  The 5 can be left out of the array size,
8     //and the compiler will automatically find its size.
9     int testScores[5] = {86, 95, 69, 96, 89};
10
11    cout << testScores[0] << endl;
12    cout << testScores[1] << endl;
13    cout << testScores[2] << endl;
14    cout << testScores[3] << endl;
15    cout << testScores[4] << endl;
16
17    return 0;
18  }
```

Figure 5.3.2

By enclosing the numbers in curly braces, and separating them with commas, and array literal is created. When declaring and initializing an array, the values can be directly placed into the array without needing to assign to each element. The size of the array is not needed when initialized this way, since the compiler can automatically calculate its size based on the assigned array literal.

## Section 5.4: Arrays and Loops

The power of arrays becomes apparent when combined with loops. By using the loop variable as the index of an array, the entire content of an array can be processed in only a few lines of code. As an example of how arrays and loops can be used to simplify a problem, consider a program designed to find the average of a set of five test scores. Using individual variables, the program will look like the code in figure 5.4.1.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Multiple declaration of test score variables
6     int testScore0, testScore1, testScore2, testScore3, testScore4;
7
8     //Assign a test score to each variable
9     testScore0 = 86;
10    testScore1 = 95;
11    testScore2 = 69;
12    testScore3 = 96;
13    testScore4 = 89;
14
15    //Calculate the average of the five scores.  What if we had 20 scores?
16    double average = (double)(testScore0 + testScore1 + testScore2 +
17                                        testScore3 + testScore4) / 5;
18
19    return 0;
20  }
```

Figure 5.4.1

While the code is straightforward for five scores, realistically there will be dozens of scores belonging to many students. In addition, the program is inflexible. If the number of test scores were to change, several pieces of the code would need to be modified in order to accommodate a new score. By using a loop and array, the code could be simplified, as shown in figure 5.4.2.

Figure 5.4.2

```
1   #include <iostream>
2   using namespace std;
3
4   const int arraySize = 5;
5
6   int main(){
7     //Initialize testScore with the test scores
8     int testScore[arraySize] = {86, 95, 69, 96, 89};
9
10    //This will contain the sum of the test scores.
11    //It is the (testScore0 + testScore1 …) part of the previous program
12    int accumulator = 0;
13
14    //This does the job of adding each test score to accumulator
15    //The loop variable takes on the values 0, 1, 2, 3, and 4 ...
16    for(int i = 0; i < arraySize; i++){
17      //...which means that accumulator = testScore[0]+ testScore[1] ...
18      accumulator += testScore[i];
```

```
19    }
20    //Finally, divide the sum of scores by the number of scores.
21    cout << "Average is " << (double)accumulator / arraySize;
22
23    return 0;
24  }
```

Figure 5.4.2 (cont.)

The loop variable generally starts at 0 since the first index of an array is 0. Arrays and loops are commonly used together, which is why the two correlate. Adding another score to this simplified program is far easier than before. The code in figure 5.4.3 does this by adding 1 to **arraySize** and adding the new number to **testScore**.

```
1   #include <iostream>
2   using namespace std;
3
4   const int arraySize = 6;
5
6   int main(){
7     //Initialize testScore with the test scores
8     int testScore[arraySize] = {86, 95, 69, 96, 89, 92};
9
10    //This will contain the sum of the test scores.
11    //It is the (testScore0 + testScore1 …) part of the previous program
12    int accumulator = 0;
13
14    //This does the job of adding each test score to accumulator
15    //The loop variable takes on the values 0, 1, 2, 3, and 4 ...
16    for(int i = 0; i < arraySize; i++){
17      //...which means that accumulator = testScore[0]+ testScore[1] ...
18      accumulator += testScore[i];
19    }
20    //Finally, divide the sum of scores by the number of scores.
21    cout << "Average is " << (double)accumulator / arraySize;
22
23    return 0;
24  }
```

Figure 5.4.3

## Section 5.5: Multi-dimensional Arrays

The type of array used in the previous examples is a one-dimensional array. A one-dimensional array is similar to a row of variables lined up together. In some programs, data is more naturally organized into tables such such as a tic-tac-toe board. In order to represent

structures like this in a program, a **two-dimensional array** can be used. A two-dimensional array functions like a table, where elements are defined by their row and column. In a similar way, a two-dimensional array uses two indices to reference an element. The program in figure 5.5.1 declares and initializes a two-dimensional array. A nested loop is then used to display the board to the screen.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare a 3x3 character array and initialize it with values.
6     char ticTacToeBoard[3][3] = {{'X', 'O', '_'},
7                                  {'O', '_', 'X'},
8                                  {'X', '_', 'O'} };
9
10    //Use nested loops to iterate over the two indexes of the array.
11    //The outer loop iterates over the rows
12    for(int row = 0; row < 3; row++){
13      //The inner loop iterates over the columns
14      for(int col = 0; col < 3; col++){
15        //Addressing an element requires two indexes
16        cout << ticTacToeBoard[row][col] << " ";
17      }
18      cout << endl;
19    }
20
21    return 0;
22  }
```

Figure 5.5.1

A two-dimensional array is declared by using a one-dimensional array declaration, and adding an additional dimension by placing square brackets and the size of the extra dimension. A two-dimensional array is initialized by assigning it an array of arrays. In this code, there are three array literals that are themselves members of an array. The outer most part of the array contains the three elements {'X', 'O', '_'}, {'O', '_', 'X'}, and {'X', '_', 'O'}. Each of these elements is itself an array.

The nested for loop displays the content of the array on the screen. The outer for loop on line 12 loops over the rows of the array while the inner loop takes the **row** variable from the outer loop and iterates over the elements in that row. When referencing an element in a two dimensional array, two indices are used in order to identify the row and column of the element within the array.

Arrays in C++ can be declared with more than two dimensions. In practice, however, one dimensional and two dimensional arrays are most frequently used while arrays of three or higher dimensions are rarely seen.

## Section 5.6: What are Strings?

When developing programs, text is commonly used. Nearly every example so far has used text into order to communicate with the user, such as displaying "Enter a number: " or "Hello, World!". In addition to interfacing with the user, data may be stored in a file in the form of text. In C++, the data type used to hold text are called **strings**. In the C++ language, there are two types of strings that can be used. The first are **C strings**, which come from the C programming language, a predecessor of C++. A C string is an array of characters, with the last character being the null byte (the character '\0'). The difference between an array of characters and a C string is the presence of the null byte. Without the null byte, it is no longer a C string. When processing a string, a loop can be used to process the content and detect the end of the string when the null byte is encountered. Another type of string exclusive to C++ is the **string type**. The string type is easier to use since it encapsulates the details of how the data is stored. Nevertheless, knowing how to use both types of strings is important.

The code in figure 5.6.1 declares both a C and C++ style string and displays their content using **cout**.

Figure 5.6.1

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declaring a cstring is just declaring
6     //a character array. The string literal "Hello, World!"
7     //implicitly ends in a null byte, which
8     //ensures that this is a string and not
9     //a character array.
10    char cString[] = "Hello, World!";
11
12    //A C++ string
13    string cppString = "Hello, World!";
14
15    //Both types of string are displayed
16    //the same way using cout.
17    cout << cString << endl;
18    cout << cppString << endl;
19
20     return 0;
21
22  }
```

It is also important to note that C string cannot be reassigned a string literal, while C++ strings can, as shown in figure 5.6.2 below.

Figure 5.6.2

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //A C string cannot be reassigned a value.
6    //Uncomment the line below to
7    //see how the program fails to compile
8    char cString[] = "first value";
9    //cString = "cannot do this";
10
11   //A C++ string can be reassigned, however
12   string cppString = "first value";
13   cppString = "can do this";
14
15   cout << cppString << endl;
16
17   return 0;
18 }
```

## Section 5.7: Processing Strings

Strings are an important data type in C++, and processing strings forms the basis for important pieces of software such as compilers. Whereas the primitive types seen so far, such as **int** and **double**, have relatively simple operations that are performed on them, there are many operations that can be performed on strings. Below are some of the most common and useful operations.

- String Length

When processing strings, the length is often used as the basis for more complicated algorithms. For example, a password checking program may require that a new password be a minimum of 8 characters in length in addition to other specifications. In order to check this requirement, the length of the password would need to be checked. The code in figure 5.7.1

Figure 5.7.1

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Initialize a C++ string
6    string cppString = "hello";
7
8    cout << "cppString length " << cppString.length() << endl;
9
10   return 0;
11 }
```

shows how to find the length of a C++ string and is contrasted to finding the length of a C string in figure 5.7.2.

Whereas the C string is simply an array of characters with a newline added to the end, a C++ string is a more complex type called an **object**. Objects will be covered in a future chapter. For now, notice how the loop that is required to find the length of a C string in the code to the right is not needed for a C++ string. Instead, the C++ string has a special **method** called length that can be used to automatically find its length without using a loop.

Figure 5.7.2

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Initialize a C string
6    char cstring[] = "hello";
7
8    //Initialize the starting length to 0
9    int cstringLength = 0;
10
11   //Loop over every character until
12   //the null byte is encountered
13   while( cstring[cstringLength] != '\0' ){
14     cstringLength++;
15   }
16
17   cout << "Length is " << cstringLength << endl;
18
19   return 0;
20 }
```

- Comparing strings

In addition to finding the length of a string, comparing two strings is also a common operation that needs to be performed. As an example, a search engine may take a string as input and attempt to match it to find search results. The process of performing this is complex, and comparing whether two strings are the same is easier. When comparing two C strings, a **function** called **strcmp** is used from the **cstring** library. Functions work similarly to how a calculator operates by accepting some input, processing the input, and returning a result. In the code for figure 5.7.3, for example, the input is *cstring* and *cstring2* and the result is a number that determines whether the two C strings are equal. Calling a function results in a value being returned. For the **strcmp** function, the return value is 0 if the two strings are equal, or a non-zero value if they are not.

Figure 5.7.3

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  int main(){
6    char cstring[] = "hello";
7    char cstring2[] = "world";
8
9    //strcmp returns 0 if the strings
10   //are the same, so check if the result is 0.
11   if( strcmp(cstring, cstring2) == 0){
12     cout << "Strings are the same!" << endl;
13   }else{
14     cout << "Strings are not the same!" << endl;
15   }
16 }
```

For C++ strings, the comparison operators from a previous chapter can be used as shown in figure 5.7.4. By now, you should recognize that using C++ strings reduces the amount of code needed in addition to being easier to understand.

Figure 5.7.4

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    string cppString1 = "hello";
6    string cppString2 = "world";
7
8    if( cppString1 == cppString2 ){
9      cout << "Strings are equal!" << endl;
10   }else{
11     cout << "Strings are not equal" << endl;
12   }
13 }
```

- User input with strings

In order to retrieve text from the user, the data can be read from **cin** and stored into a string. For both C strings and C++ strings, the stream extraction operator (“>>”) can be used. The code in figure 5.7.5 and 5.7.6 show how to perform this using a C and C++ string, respectively.

Figure 5.7.5

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //This will store the input
6    char input[40];
7
8    //Prompt the user for input
9    cout << "Enter text: ";
10   cin >> input;
11
12   //Display what the user entered
13   cout << input << endl;
14
15   return 0;
16 }
```

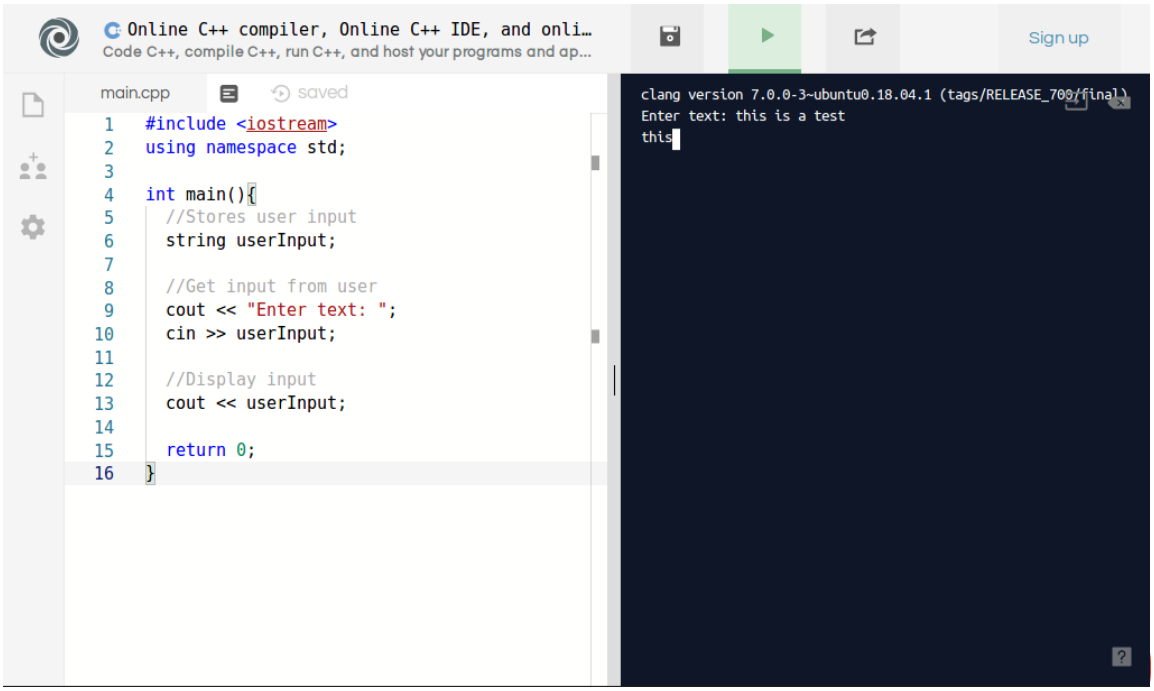
Figure 5.7.6

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Stores user input
6    string userInput;
7
8    //Get input from user
9    cout << "Enter text: ";
10   cin >> userInput;
11
12   //Display input
13   cout << userInput;
14
15   return 0;
16 }
```

When reading data into a C string, the character array must be declared and given a size. In this example, the string length is set to be 40 characters, including the required null byte. If the user entered more than this number of characters, the program would not function correctly. For the C++ version, however, the string is able to dynamically resize itself to accept any length

of input from the user. C string are fixed and static, whereas C++ strings are dynamic and can change.

In spite of this, however, both versions do not behave as expected. When text that includes spaces is entered, only the characters up until the first space is read.




This happens because the stream extraction operator delimits on spaces. When a space is encountered, no more text is read. To circumvent this problem, the **getline** function can be used. By default, **getline** delimits on the newline character ('\n'), which marks the end of the line. By delimiting on a newline, the entire line of text is read in. The code below uses the **getline** function to read in user input instead of using the stream extraction operator.

Figure 5.7.8

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    char userInput[40];
6
7    cout << "Enter text: ";
8    cin.getline(userInput, 40);
9
10   cout << userInput;
11
12   return 0;
13 }
```

Figure 5.7.9

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    string userInput;
6
7    cout << "Enter text: ";
8    getline(cin, userInput);
9
10   cout << userInput;
11
12   return 0;
13 }
```

The **getline** function has two different forms that are used, depending on if a C string or a C++ string is being used to store the input.

- Processing with a Loop

Looping over a string is often used as the basis for other string processing functions, such as counting how many time a certain character occurs in a string or searching for a specific substring. Both C and C++ strings act like arrays that can be iterated over using a loop. The code below illustrates how a loop can be used to loop over a string to count the number of times the letter "l" appears in the string.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    char myString[]= "hello, world!";
6    int numberLs = 0;
7
8    int stringIndex = 0;
9    //Loop through the string as long
10   //as the current character is not null
11   while( myString[stringIndex] != '\0'){
12     //If the current character is an
13     //'l', increment numberLs
14     if( myString[stringIndex] == 'l' ){
15       numberLs += 1;
16     }
17     stringIndex++;
18   }
19
20   cout << numberLs << " l's found" << endl;
21   return 0;
22 }
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    string myString = "hello, world!";
6    int numberLs = 0;
7
8    int stringIndex = 0;
9    //Loop through the string as long
10   //as the current character is not null
11   while( myString[stringIndex] != '\0'){
12     //If the current character is an
13     //'l', increment numberLs
14     if( myString[stringIndex] == 'l' ){
15       numberLs += 1;
16     }
17     stringIndex++;
18   }
19
20   cout<<numberLs<<" l's found"<< endl;
21
22   return 0;
23 }
```

As can be seen in these examples, C++ strings tend to be easier to work with and are preferred over C strings. Nevertheless, it is important to be able to use both forms of strings.

# Chapter 6: Functions

Chapter Layout



## Section 6.1: Why are Functions Needed?

As the size of a program increases, the same functionality will inevitably be duplicated. The code for finding the length of a string, for example, is commonly used when handling strings. To find the length of three strings, the code and associated variables would need to be duplicated three times as shown in figure 6.1.1 below.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare and initialize three strings
6     char myString1[] = “hello, world!”;
7     char myString2[] = “second string”;
8     char myString3[] = “third string”;
9
10    //Find the length of the first string
11    int stringLength1 = 0;
12    while( myString1[stringLength1] != ‘\0’ ){
13      stringLength1++;
14    }
15
16    //Find the length of the second string
17    int stringLength2 = 0;
18    while( myString2[stringLength2] != ‘\0’ ){
19      stringLength2++;
20    }
21
22    //Find the length of the third string
23    int stringLength3 = 0;
24    while( myString3[stringLength3] != ‘\0’ ){
25      stringLength3++;
26    }
```

Figure 6.1.1

```
27    //Print out the three string lengths
28    cout << "String 1 has a length of " << stringLength1 << endl;
29    cout << "String 2 has a length of " << stringLength2 << endl;
30    cout << "String 3 has a length of " << stringLength3 << endl;
31
32    return 0;
33  }
```

Figure 6.1.1 (cont.)

Copying, pasting, and changing code this way is repetitive and prone to errors. When changing the names of the **stringLength** variable, it is easy to have **stringLength1** mistakenly replaced with **stringLength3**. Worst still, if the copied code contained a logic error, every instance of the code would need to be found and corrected by hand. In a real-world application, functionality is often used dozens or even hundreds of times. Clearly, this method does not work and needs a solution.

## Section 6.2: What are Functions?

Functions solve this problem by grouping together and isolating pieces of code. Instead of copying and pasting code to be used multiple times, a function allows programmers to write the code once, and then re-use it many times in a program. This eliminates copy-and-paste, in addition to making the code easier to maintain. If a function contains a logic error, the code within the function can be fixed and all other code which uses the function will then work properly.

Functions alter the control flow of the program. When a function is called, the program jumps to the function and begins executing code. Once the functions has finished, the control flow returns back where it started. The diagram in figure 6.2.1 below shows the control flow when a function is called.

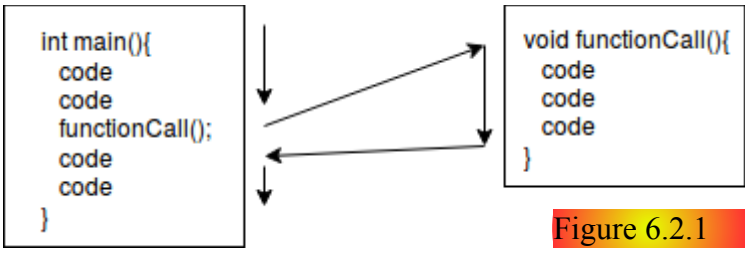



Figure 6.2.1

Functions allow better organization in a program by grouping together and isolating related functionality. By doing this, the code is modular and easier to work with. Functions also allow the inputs and outputs of code to be controlled, which provides isolate between parts of a program.

Isolating code into functions is also the basis of libraries and modules, which are important when developing software. Functions allow code to be **abstracted** by allowing programmers to focus on the high-level design of the program instead of getting bogged down in the details of the code implementation. The concept of abstracting functionality stretches far beyond the concepts covered in this book, and is an important concept in computer science. As you develop your skills as a programmer, the organization and abstraction provided by functions will increase your productivity and make it easier to create larger and more sophisticated programs.

## Section 6.3: Defining and Using Functions

Functions, just like variables, must be declared and defined in order to be used. In contrast to variables, which just have an associated type, functions have three main components that must be specified: the **parameters**, the **function body**, and the **return type**. At a high-level view, functions accept input through the parameters, process the parameters inside of the function body, and return a result based on the return type of the function. The code for figure 6.3.1 defines a function called **addNumbers**. The parameters for this function are enclosed in parenthesis that follow the function name. In this case, the two parameters are of type **int** and are called **number1** and **number2**, respectively. The code inside of the function processes the parameters by adding them and storing the result in a variable named **result**. The function returns the result by using the return statement on line 14. By using the return statement, the code that calls the function can receive the processed information.

```
1   #include <iostream>
2   using namespace std;
3
4   //Declare a function named addNumbers.
5   //This function will accept two parameters
6   //(number1 and number2) both of type int.
7   //The function will return a value of type int.
8   int addNumbers(int number1, int number2){
9     //Add the two parameters and assign the
10    //value to result
11    int result = number1 + number2;
12
13    //Return the value of result.
14    return result;
15  }
16
17  int main(){
18    //Call the function addNumbers with the
19    //arguments 1 and 1.
20    //Store the returned value into additionResult
21    int additionResult = addNumbers(1, 1);
22
23    //Print out the result
24    cout << "Result is " << additionResult << endl;
25
26    //The return value can be directly displayed
27    cout << addNumbers(2, 2) << endl;
28
29    return 0;
30  }
```

Figure 6.3.1

The type that is being returned must match the return type specified at the beginning of the function on line 8. Once the value is returned, it can be used almost anywhere in a program, such as assignment statements, mathematical expressions, and boolean expressions. Line 8 in figure 6.3.1 is called the **function header**, which consists of the return type, name, and parameters for the function.

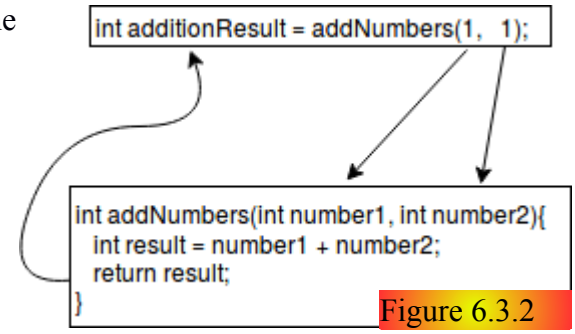


Figure 6.3.2

The function is **called** by writing the function name followed by the arguments for the function. Line 21 of figure 6.3.1 calls the **addNumbers** function by specifying the value of 1 for the parameter **number1** and the value 2 for the parameter **number2**. When values are passed to a function in this manner, they are called **arguments**. The parameters and arguments are positional, which means that the first argument in the function call will be assigned to the first parameter in the function header. A function can be called with different arguments to process different pieces of data. The diagram in figure 6.3.2 shows how the function call and the function itself are related.

```
1  #include <iostream>
2  using namespace std;
3
4  //Declare a function named stringLength that accepts a string
5  //and returns its length as an integer
6  int stringLength(char stringInput[]){
7    //Set initial length to 0
8    int length = 0;
9
10   //Loop through the string, incrementing length, until
11   //the null byte has been hit.
12   while( stringInput[length] != '\0' ){
13     length++;
14   }
15
16   //Return the length of the string
17   return length;
18 }
19
20 int main(){
21   //Declare and initialize three strings
22   char myString1[] = "hello, world!";
23   char myString2[] = "second string";
24   char myString3[] = "third string";
25
26   //Print out the three string lengths. Notice how we can use the
27   //stringLength function directly in these expressions.
28   cout << "myString1 length: " << stringLength(myString1) << endl;
29   cout << "myString2 length: " << stringLength(myString2) << endl;
30   cout << "myString3 length: " << stringLength(myString3) << endl;
31
32   return 0;
33 }
```
Figure 6.3.3

By using a function instead of duplicating code, the problem of duplicating the

string length code in the opening of this chapter can be solved. When turned into a function, the **stringLength** procedure can be written once and be used to find the length of multiple strings by calling the function with different arguments. With this approach, a logic error in the **stringLength** code could be fixed once, and all calls to the function would use the fixed code.

Finally, a function may not always need to return a value. For example, the function in figure 6.3.4 displays the text "Hello!" and does not need to return a value. In order to specify that no value is returned from a function, the return type is specified as **void**. When a function's return type is **void**, no return statement is needed within the function body.

```
1  #include <iostream>
2  using namespace std;
3
4  //The return type is void since nothing is returned.  Notice also
5  //there are no parameters.  This function does not take any.
6  void sayHello(){
7     cout << "Hello!" << endl;
8     //No return statement since the return type is void
9  }
10
11 int main(){
12    //This calls the sayHello function.  Since it returns nothing,
13    //we can't have a variable be assigned its value.
14    sayHello();
15    return 0;
16 }
```

Figure 6.3.4

## Section 6.4: Function Prototypes

If the function **sayHello** from figure 6.3.4 were moved after the main function, a problem would occur. The code in figure 6.4.1 moves **sayHello** to the bottom, and an error results.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5     //Call the sayHello function.
6     sayHello();
7
8     return 0;
9  }
10
11 //The return type is void since nothing is returned.  Notice also
12 //there are not parameters.  The function does not take any.
13 void sayHello(){
14    cout << "Hello!" << endl;
15    //No return statement since the return type is void
16 }
```

Figure 6.4.1

When the compiler transforms the source code into an executable, it scans from the first line of code in the file to the last, just as a human would read a book. Because of this, the compiler first encounters the function being called on line 6 of figure 6.4.1 before the definition on lines 13 through 16. Just as a human would be confused if a character suddenly appeared within the plot of a book, the compiler will not recognize **sayHello**.

To correct this, the **function prototype** for **sayHello** needs to be placed before the function is called in main. Figure 6.4.2 provides the function prototype on line 7. The function prototype includes the return type, the name, and the parameter list for the function and is exactly the same as the function header. Just as a character must first be introduced in a book, the function prototype introduces the **sayHello** function so that it can be called in the main function.

Figure 6.4.2

```
1   #include <iostream>
2   using namespace std;
3
4   //The function prototype.  This has the
5   //function return type, the function name,
6   //and the parameters.
7   void sayHello();
8
9   int main(){
10    //Call the sayHello function
11    sayHello();
12    return 0;
13  }
14
15  //The return type is void since nothing is returned.
16  //Notice also how there are no parameters specified.
17  //The function does not accept any.
18  void sayHello(){
19    cout << "Hello!" << endl;
20    //No return statement since the return type is void
21  }
```

## Section 6.5: Recursion

A general strategy of solving a problem is to break the overall problem into a series of simpler problems, and then solve each of the simpler problems. The sub-problems may need to be broken down further into even simpler problems. Once all of the sub-problems have been solved, the results are combined to form an overall solution to the original problem.

Recursion operates this way by breaking a problem into a simpler version of itself until a trivial problem remains, at which point the trivial problem is easily solved. The sub-problems are then combined to form a solution to the overall problem at hand. In the **recursive case,** the problem is broken into simpler versions of itself until a **base case** is reached, at which point the problem can be automatically solved.

As an example of how recursion may be used, consider the factorial function. The factorial of a number **n** may be reduced to a simpler version of the same problem by multiplying **n** by the factorial of **n – 1**. This process continues until the base case of 1 has been reached, at which point the result is automatically 1. The diagram in figure 6.5.1 shows how a recursive factorial function can calculate the factorial of the number 5.

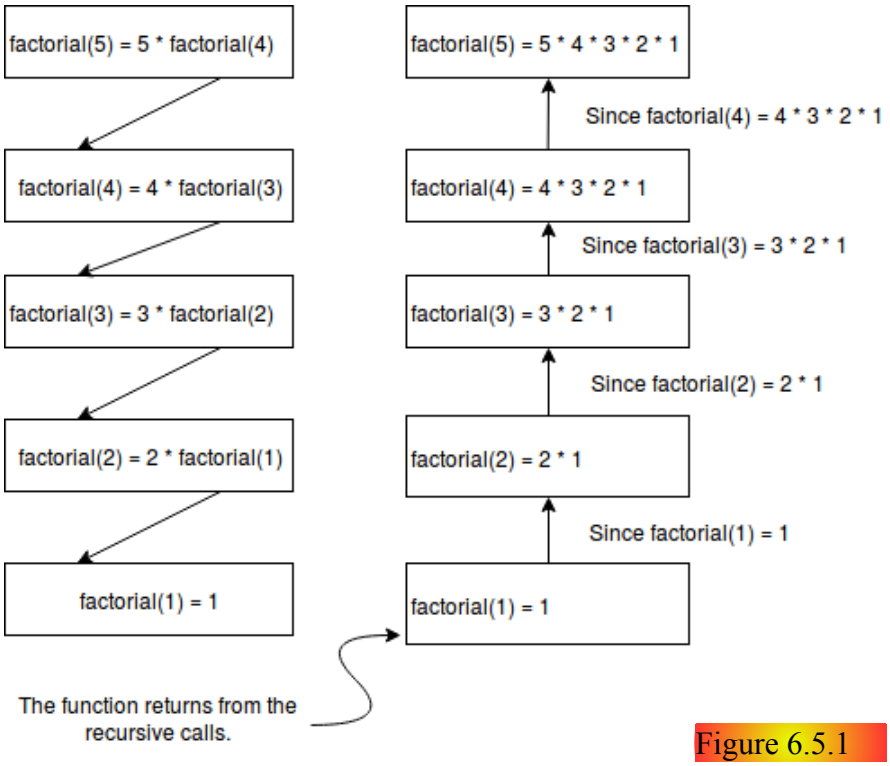



Figure 6.5.1

The code in figure 6.5.2 is a recursive implementation of the factorial function illustrated in the diagram. The if statement on line 6 checks if the base case of 1 has been reached. If the factorial function is passed the value of 1, the result is automatically 1. Alternatively, if a number greater than 1 has been passed into the function, the result is simplified as the number times the factorial of the number minus 1.

```
1  #include <iostream>
2  using namespace std;
3
4  //Recursive factorial function
5  int factorial(int number){
6      if( number == 1 ){
7        //Base case. factorial(1) = 1 by definition
8        return 1;
9      }else{
10       //Recursive case. multiple the number by
11       //the factorial of the number minus 1
12       return number * factorial(number-1);
13     }
14 }
15
16 int main(){
17   cout << "Final result: " << factorial(5) << endl;
18   return 0;
19 }
```

Figure 6.5.2

# Chapter 7: Sorting and Searching

Chapter Layout



## Section 7.1: Why Sorting and Searching?

At the core of processing data is sorting and searching. Sorting arranges data into some useful order that can be used to process the information. Searching retrieves data based on a certain criteria within a larger body of data. An online multi-player game, for example, may sort thousands of users by rank to find the top player, or search for a specific user by name. In order to sort and search through data, a specific **algorithm**, which is a set of well-defined steps that lead to some goal, are implemented within the program.

A **sorting algorithm**, for example, defines the steps needed to transform an unsorted set of numbers into sorted order. Sorted order for an array of numbers could mean sorting in ascending or descending order. For more complex types of data, such as user profiles, a specific part of the profile could be sorted on, such as the age of the individual. The three types of sorting algorithms explored in this chapter are **bubble sort**, **selection sort**, and **insertion sort**.

A **search algorithm** defines the steps to follow in order to find a specific piece of data from a larger set of information. A search algorithm will use a criteria, such as a unique ID or a user name, and search through the provided information to find a match. The two search algorithms presented in this chapter are **linear search** and **binary search**.

Algorithms are an important topic in computer science as they represent the building blocks for larger programs. An algorithm in the most general sense is a set of steps to solves a particular problem. In order to obtain a degree from a university, for example, you would need to understand what courses to take, when to take them, and how to complete each course. These processes are all components of a larger algorithm that leads to the end goal of receiving a degree.

## Section 7.2: Bubble Sort

The **bubble sort** algorithm sorts data by comparing each pair of elements to put them in sorted order. The illustration in figure 7.2.1 to the right uses an unsorted array of numbers and proceeds to sort the numbers in ascending order using the bubble sort algorithm.

The algorithm begins by comparing the first and second elements of the array. To order the first two elements correctly, the number 3 would need to come before the number 4. Because of this, the two elements are swapped so that they are in sorted order.

In the second step, the next pair of elements are compared. Because the numbers should be in ascending order, and since the number 1 should come before the number 4, the two elements pointed to are swapped.

The next pair of elements are compared. Since 4 comes before 8, the numbers are in the correct order and no swapping occurs.

Finally, the last pair of numbers are checked. Since the number 7 should come before the number 8, a swap occurs.

The entire process shown in the diagram is called a **pass** over the array. A single pass will not necessarily completely sort the array, as shown in the final ordering of the numbers in the figure. The bubble sort algorithm will make

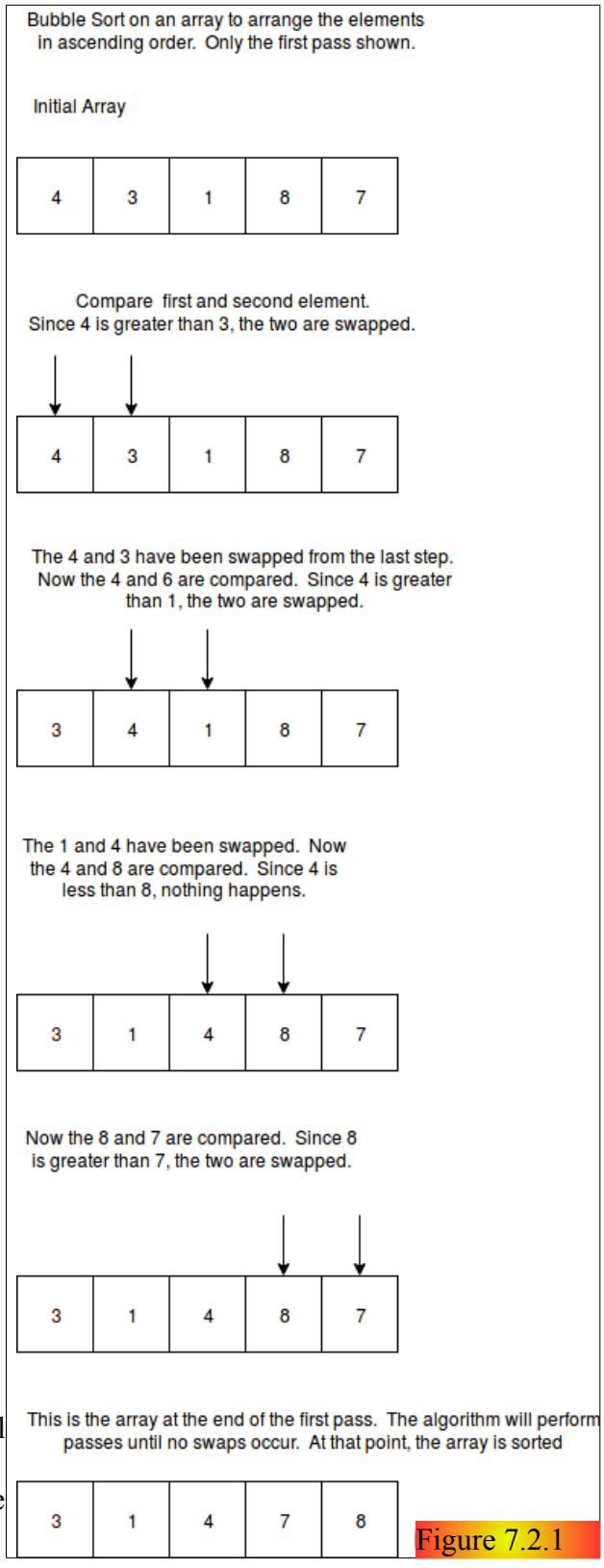


Figure 7.2.1

passes through the array, which will cause the data to become closer to sorted order after every pass. When a full pass has been executed and no swapping occurs, the array is then known to be sorted and the bubble sort algorithm stops. The code in figure 7.2.2 below implements the bubble sort algorithm as a function and uses it to sort an array of numbers.

```
1   #include <iostream>
2   using namespace std;
3
4   //Bubble sort an integer array in ascending order
5   void bubblesort(int myArray[], int arraySize){
6     //Once no swap occurs over a complete pass,
7     //we know the array is sorted.
8     bool swapOccurred = true;
9
10    //Continue to make passes on the array while
11    //a swap occurred
12    while(swapOccurred){
13      //Initially, no swapping has happened
14      swapOccurred = false;
15
16      //Perform an individual pass on the array
17
18      //This is the index of the first
19      //element to compare
20      int firstIndex = 0;
21
22      //This is the index of the second element
23      //to compare
24      int secondIndex = 1;
25
26      //Continue to loop over the array until
27      //we hit the end of the array
28      while( secondIndex < arraySize ){
29
30        //This sorts in ascending order, so if
31        //a larger number comes before a smaller
32        //number, then they need to be swapped
33        //The greater than comparison can be changed
34        //to less than to sort in descending order.  Once you
35        //get the code working, try changing this.
36        if( myArray[firstIndex] > myArray[secondIndex] ){
37
38          //Swap the two elements
39          int temp = myArray[secondIndex];
40          myArray[secondIndex] = myArray[firstIndex];
41          myArray[firstIndex] = temp;
```

Figure 7.2.2

```
42
43          //Make sure we indicate that a swap
44          //has happened
45          swapOccurred = true;
46        }
47
48        //Move to the next two elements
49        firstIndex++;
50        secondIndex++;
51       }
52     }
53   }
54
55  int main() {
56    int a[] = {4,3,1,8,7};
57    int aSize = 5;
58    bubblesort(a, aSize);
59
60    cout << "After sorting: ";
61    for(int i = 0; i < aSize; i++){
62      cout << a[i] << " ";
63    }
64    cout << endl;
65  }
```

Figure 7.2.2 (cont.)

## Section 7.3: Selection Sort

If you were asked to sort a pile of unsorted numbers, one strategy that could be used is to begin by placing the smallest number into its own pile. In the second step, the next smallest number is chosen and lined up with the first number. In the third step, the next smallest number is placed into the sorted pile. This process continues until there are no unsorted blocks remaining. The diagram in figure 7.3.1 to the right illustrates how this process happens. Because the lowest block is always chosen, it will always be placed at the end of the sorted numbers.

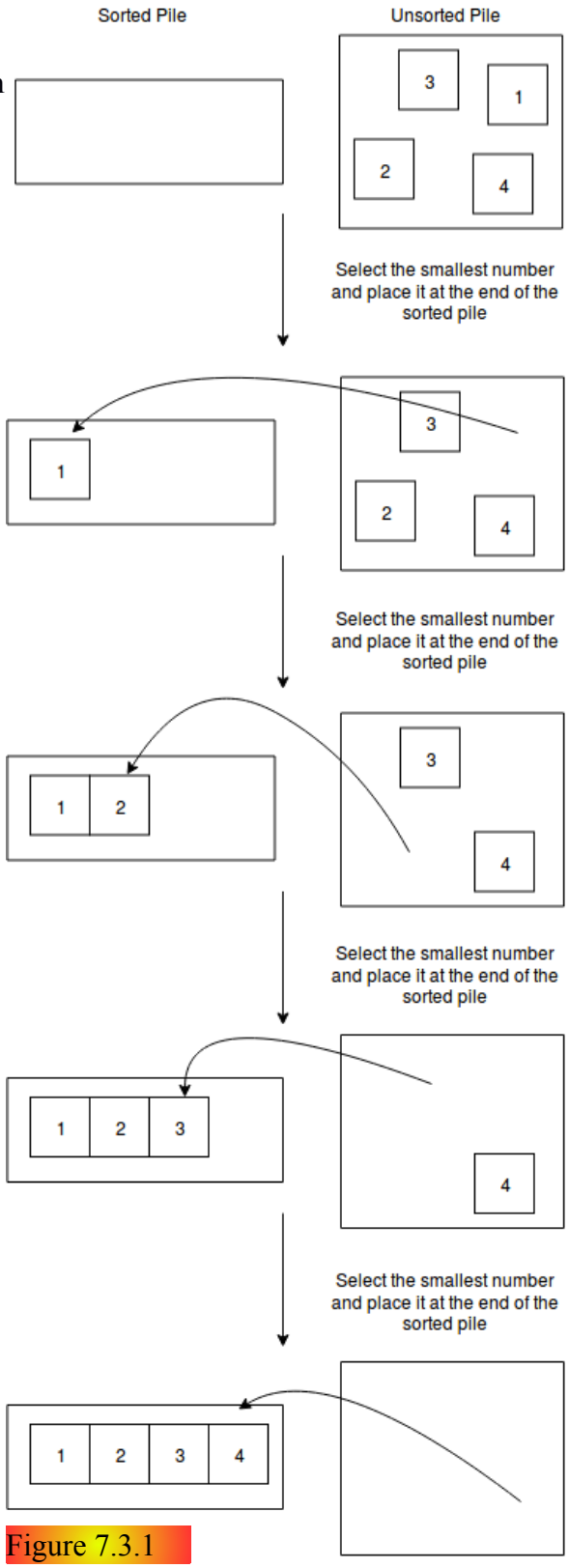



Figure 7.3.1

The selection sort algorithm works in a similar way by logically separating the array into a sorted and unsorted portion. The sorted portion consists of all elements to the left of the partition, not including the element at the index of partition. The unsorted elements are to the right of the partition, including the element where partition is currently placed.

Initially, all of the numbers are considered unsorted, so the partition contains index 0. To begin, the smallest element within the unsorted portion is found and swapped with the number under partition. In this example, the number 1 is the smallest value in the array, so it is swapped with the number under partition, which is 4. After the numbers have been swapped, the partition moves to the next index in the array.

At this point in the algorithm, the sorted portion of the array is the first element, while the unsorted part are the elements from the partition to the end of the array. The algorithm continues by searching for the smallest element in the unsorted portion. Since 3 is the smallest and is already in the correct location, the number does not move.

The partition moves to the next index in the array. The next smallest number in the array is 4. As in the last case, the 4 is in the correct location and is not moved.

The partition moves to the next index, and finds that the smallest number in the unsorted portion is the 7. This number is swapped with the number 8 under the partition.

Finally, the algorithm reaches the last element in the array. Since there is only one number left, it is automatically the smallest and in the correct location. The array is now in sorted order and the selection sort algorithm terminates.

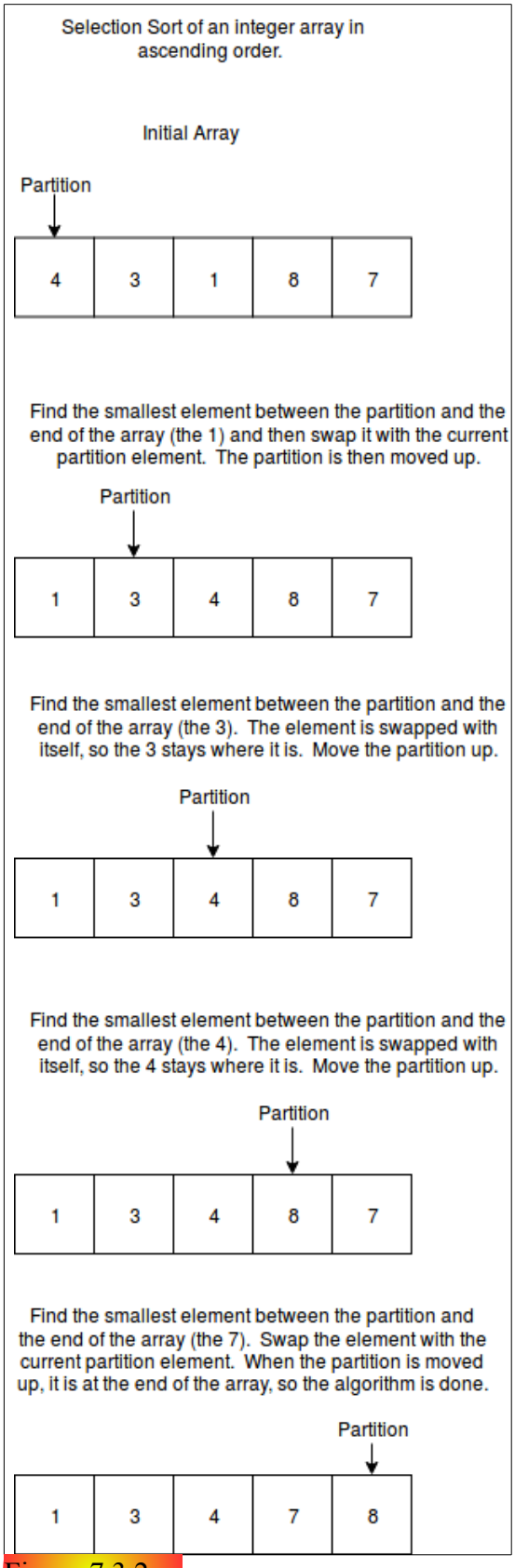


Figure 7.3.2

```
1   #include <iostream>
2   using namespace std;
3
4   //Selection sort in ascending order
5   void selectionsort(int myArray[], int arraySize){
6
7     //Initially, the partition will begin at the first element
8     //which means that every element is in the unsorted portion.
9     int partition = 0;
10
11    //Continue to move the partition until it is at the
12    //end of the array.  Once this happens, all of the elements
13    //elements will be in the sorted portion.
14    while(partition < arraySize){
15      //Scan the unsorted numbers to select the
16      //smallest element.
17      int smallestNumber = myArray[partition];
18      int smallestNumberIndex = partition;
19      for(int i = partition; i < arraySize; i++){
20        if( myArray[i] < smallestNumber ){
21          smallestNumber = myArray[i];
22           smallestNumberIndex = i;
23        }
24      }
25
26      //Now swap the smallest element found
27      //with the current partition index.
28      int temp = myArray[partition];
29      myArray[partition] = myArray[smallestNumberIndex];
30      myArray[smallestNumberIndex] = temp;
31
32      //Move partition up by 1 and continue
33      partition++;
34    }
35  }
36
37  int main() {
38    //The array to be sorted
39    int a[] = {3,4,2,6,1,3,5};
40    int asize = 7;
41    selectionsort(a, asize);
42
43    cout << "After sorting: ";
44    for(int i = 0; i < asize; i++){
45      cout << a[i] << " ";
46    }
```

Figure 7.3.3

```
47    cout << endl;
48
49    return 0;
50  }
```

Figure 7.3.3 (cont.)

## Section 7.4: Insertion Sort

Another strategy that can be used to sort a group of unordered numbers is to begin with an empty sorted pile and insert the unsorted numbers one at a time. As the unsorted numbers are added in, they are placed in sorted order.

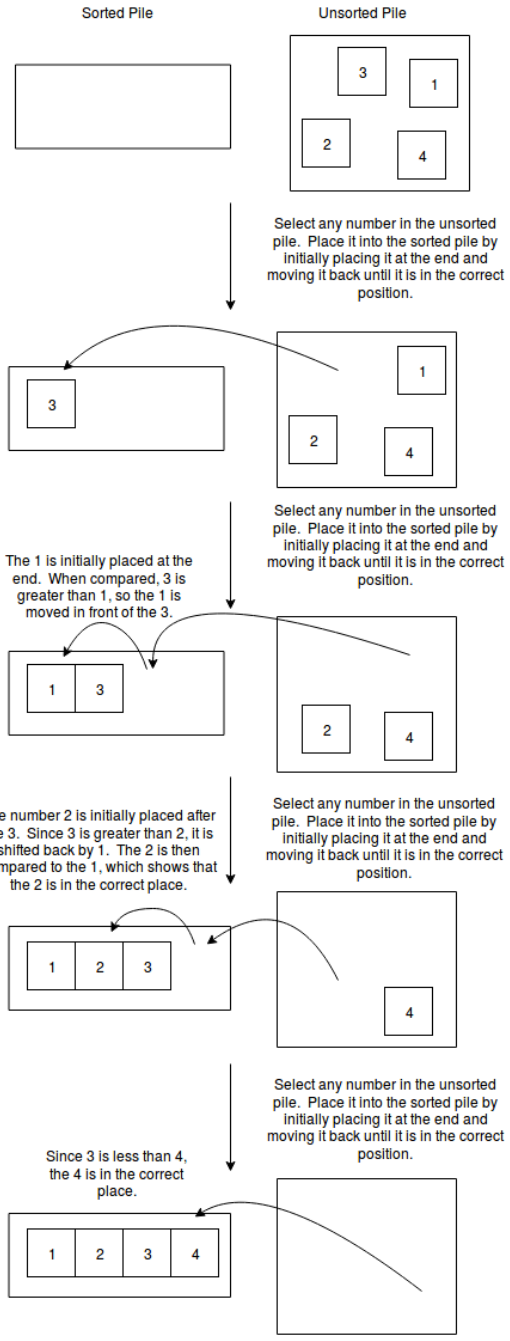
For the first step, any number is chosen to be placed into the sorted pile. The number chosen does not matter. Because there is only one number in the sorted pile, by definition it is sorted.

For the second step, another number is chosen from the unsorted pile and placed at the end of the sorted numbers. In the illustration, the number 1 is chosen and placed after the 3. The number 1 is then move backward in the sorted list until it is in the correct place.

Another number is chosen from the unsorted numbers and placed at the end of the sorted numbers. In this example, the number 2 was chosen and placed after the 3. The number 2 is then moved backward in the sorted pile until it is in the correct location.

Finally, the last number is inserted at the end of the sorted numbers. The number 4 is in the correct location and does not move. Since there are no numbers remaining in the unsorted pile, the algorithm terminates and the numbers have been sorted.

Figure 7.4.1



The selection sort algorithm works in a similar way. Initially, the array is logically divided into a sorted and unsorted portion just like the selection sort algorithm.

To begin, the partition is initially at 0 since all of the numbers are unsorted.

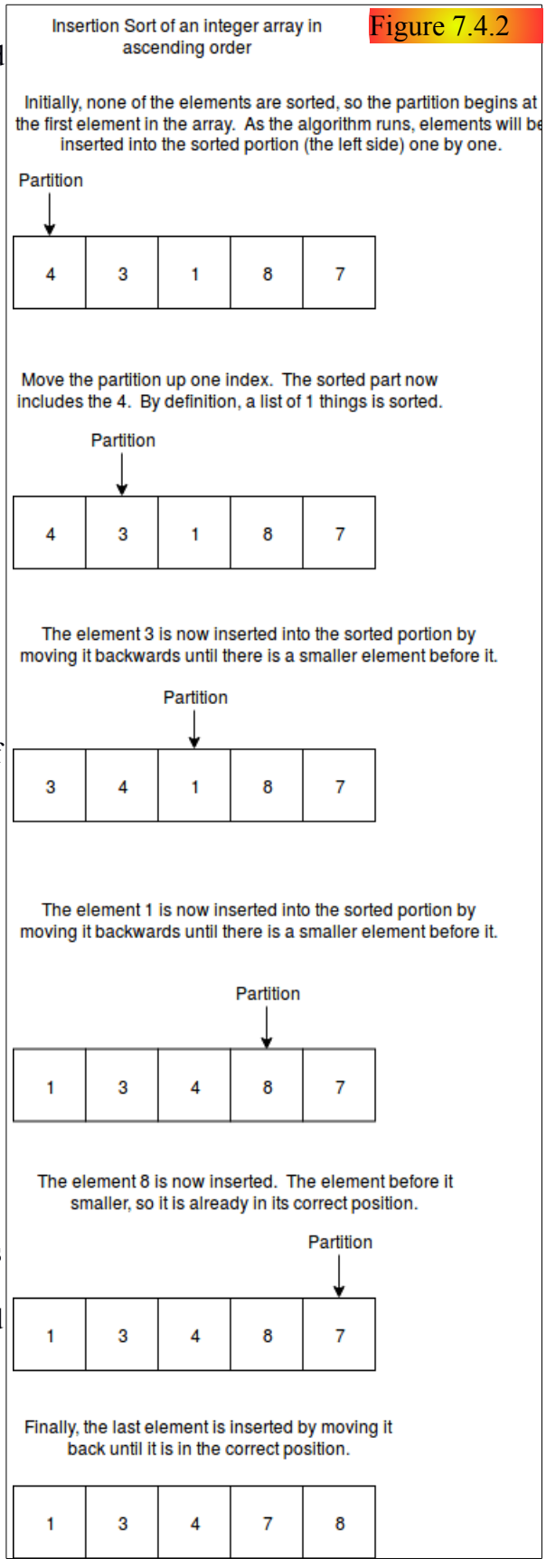
In the second step, the partition in moved up by 1 to place the first number into the sorted list. By definition, a list that has one number is sorted.

The algorithm then takes the number 3 and inserts the number into the sorted portion of the array. This is done by moving the number 3 backwards in the array until it is in the correct location. Since the 3 should come before the 4, the numbers are swapped.

Once the number 3 has been correctly placed, the partition moves to the next index. The number 1 is then inserted into the sorted portion of the array by moving it backwards until it comes before the 3 and 4.

The partition moves to the next element in the array. Since the number 8 is already in the correct location, it is not moved.

Finally, the index reaches the last number in the array. The number 7 is placed into the sorted portion of the array by moving it backwards until it is in the correct location. Since the 7 should come before the 8, but after the 4, the 8 and 7 are swapped. Because the last index of the array has been reached, the insertion sort algorithm terminates and the array is in sorted order. The code in figure 7.4.3 below implements insertion sort using as a function.



```
1  #include <iostream>
2  using namespace std;
3
4  //Insertion sort in ascending order
5  void insertionsort(int myArray[], int arraySize){
6    //Just like selection sort, start the partition
7    //at the beginning of the array.
8    int partition = 0;
9
10   //While the partition has not reached the
11   //end of the array, continue.
12   while(partition < arraySize){
13     //Insert the element at the current partition
14     //index into the sorted portion of the array.
15     int insertedElementIndex = partition;
16
17     //Move the element back until there is a smaller element before it
18     while( insertedElementIndex > 0 &&
19           myArray[insertedElementIndex-1] > myArray[insertedElementIndex]){
20
21       //Move the element back
22       int temp = myArray[insertedElementIndex-1];
23       myArray[insertedElementIndex-1] = myArray[insertedElementIndex];
24       myArray[insertedElementIndex] = temp;
25
26       insertedElementIndex--;
27     }
28
29     //Move onto the next index
30     partition++;
31   }
32 }
33
34 int main() {
35   //The array to sort
36   int a[] = {3,4,2,6,1,3,5};
37   int asize = 7;
38   insertionsort(a, asize);
39
40   cout << "After sorting: ";
41   for(int i = 0; i < asize; i++){
42     cout << a[i] << " ";
43   }
44   cout << endl;
45   return 0;
46 }
```

Figure 7.4.3

## Section 7.5: Linear Search

Linear search is the simplest, although one of the least efficient, forms of searching. A linear search starts at the first element in the array and checks if it matches the criteria. If it does, then the data is returned and the algorithm stops. If the data did not match, the second piece of data is checked, then the third, fourth, fifth, and so on. Either the data is found or the data is not present, and the end of the array is reached. Because the algorithm will compare every piece of data if necessary, it is able to work on any set of data regardless if it is sorted or not. The code in figure 7.5.1 implements a linear search function over an array of integers.

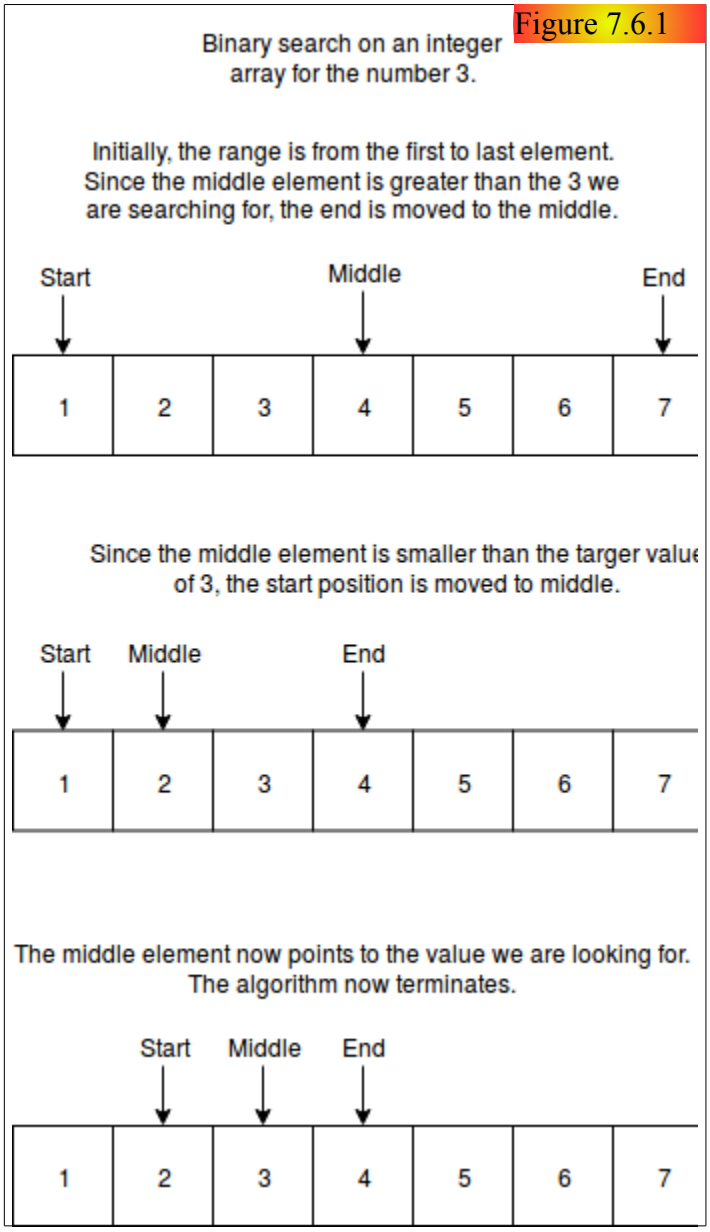
```
1   #include <iostream>
2   using namespace std;
3
4   //Linear search through an integer array
5   int linearSearch(int myArray[], int arraySize, int element){
6
7     //Loop over each element in the array and
8     //check if it is the correct one
9     for(int i = 0; i < arraySize; i++){
10      //If the element has been found, return its index
11      if(myArray[i] == element){
12        return i;
13      }
14    }
15    //Element not found. Return -1 as an error value.
16    return -1;
17  }
18
19  int main(){
20    //The array to search
21    int a[] = {4,2,3,5,7,9,6,8};
22    int aSize = 8;
23    cout << linearSearch(a, aSize, 5) << endl;
24  }
```

Figure 7.5.1

## Section 7.6: Binary Search

While simple, linear search is very inefficient since every element may need to be checked. From databases to search engines, quickly searching for data is a necessity and needs to be fast. If the data is in sorted order, a **binary search** may be performed. A binary search is able to find data faster than a linear search by continually halving the amount of data being considered.

Figure 7.6.1




Binary search is a strategy that could be used if you were to play a game where you had to guess a number between 0 and 100. Instead of guessing 0, then 1, then 2, and so on, a more efficient strategy would be to guess 50 and determine if the guess was too high or low. If the guess was too high, then the correct number must be between 0 and 49. This range may be halved again by choosing 25. If that number is too small, then the range becomes 26 to 49. This process is repeated until the number is found. Because of this, binary search works faster than linear search by eliminating more elements per guess. The code in figure 7.6.2 below implements binary search using a loop. In figure 7.6.3, recursion is used to perform a binary search. These two methods of implementing the same algorithms are called the **iterative approach**, which uses a loop, and the **recursive approach**, which uses recursion. Again, this shows how there are multiple correct ways of performing the same task in programming. The difference in this case is the efficiency and if the data is in sorted order or not. As a programmer, you would have to decide based on the use case if a linear search or a binary search is more appropriate for the task at hand.

```
1  #include <iostream>
2  using namespace std;
3  //Binary search using the iterative approach
4  int binarySearch(int myArray[], int aSize, int item){
5    //Initially, the binary search will look
6    //over the whole array
7    int start = 0;
8    int end = aSize;
9
10   //So long as the range that binary search
11   //has is not 0, continue on
12   while( start < end ){
13     //Find the midpoint of the range
14     int midpoint = (start + end) / 2;
15
16     //If the midpoint has data large than the
17     //one we are searching for, then the
18    //number we are looking for must be between
19     //the start and midpoint
20     if( myArray[midpoint] > item){
21       end = midpoint;
22
23     //If the midpoint has data less than what
24     //we are searching for, then the number
25     //we are looking for must be between the
26     //midpoint and the end.
27     }else if( myArray[midpoint] < item ){
28       start = midpoint;
29     //In this case, the midpoint must be
30     //equal to the item, so return its index.
31     }else{
32       return midpoint;
33     }
34   }
35
36   //Number is not in the array
37   return -1;
38 }
39
40 int main() {
41   //The array to search. Binary search
42   //requires a sorted array
43   int a[] = {1,2,3,4,5,6,7,8,9};
44   int aSize = 9;
45   cout << binarySearch(a, aSize, 5);
46 }
```

Figure 7.6.2

```
1   #include <iostream>
2   using namespace std;
3
4   //Binary search using the recursive approach
5   int recursiveBinarySearch(int a[], int start, int end, int item){
6
7     if( start <= end ){
8       int middle = (start + end) / 2;
9       if(a[middle] == item){
10        return middle;
11      }else if( a[middle] > item ){
12        return recursiveBinarySearch(a, start, middle-1, item);
13      }else{
14        return recursiveBinarySearch(a, middle+1, end, item);
15      }
16    }else{
17      return -1;
18    }
19  }
20
21  int main(){
22    int a[] = {1,2,3,4,5,6,7,8,9};
23    int size = 9;
24    cout << recursiveBinarySearch(a, 0, 9, 10);
25  }
```

Figure 7.6.3

# Chapter 8: Pointers and Heap Memory

Chapter Layout



## Section 8.1: What are Dynamic Memory and Pointers?

In order to read several pieces of data from a user, an array of a fixed size could be declared and the values read in one by one.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //A maximum of 3 integers can be entered
6     int userInputs[3];
7
8     //Loop three times to read in values
9     for(int i = 0; i < 3; i++){
10
11      //Prompt the user for a number and store it
12      //into the array
13      cout << "Enter number for index " << i << " :";
14      cin >> userInputs[i];
15    }
16    return 0;
17  }
```

Figure 8.1.1

The problem with this code, however, is that the user cannot enter more than three numbers because the array is given a size of 3. Once an array has been assigned a size, it cannot change as the program runs. A solution to this problem is to declare an array so large that it will always have enough capacity. The drawback to this approach is that memory is needlessly wasted by using such a large array. Even if only part of the array is used by the program, the entire array must be reserved and cannot be used for other purposes. In addition, allocating a large array and assuming it will be large enough is imprecise and does not work well in real world applications.

A far better solution to this problem is to dynamically request a certain amount of memory as the need arises. To do this, a section of memory called **the heap** is set aside for a program to use. Initially, all of the memory within the heap is free and can be used for any purpose. As memory is needed, it can be **allocated** by the program and then **freed** when it is no longer needed. The heap works in a similar way to how roller skate rentals operate at a roller rink. As customers come to use the rink, skates are rented. Once the skates are done being used, they are returned and can be used by the next customer. Also, customers must return the skates once they are done using them, else there will be none for future customers. In exactly the same way, the heap holds unused memory which is given out as the program requests more memory. If memory is not returned to the heap, there will be none for future requests. In order to use dynamic memory, however, pointers need to be understood.

## Section 8.2: Pointers and Reference Variables

Pointers are a challenging topic for new programmers. In order to have a good understanding of pointers, consider this analogy. If you were handed an index card and asked the address of the grocery store, you could write the address on the index card. The address on the index card is the location of the store, and the index card holds the address. The address can then be referenced to find the grocery store.

Pointers are analogous to the index card, and are used to store the memory address of a variable in memory.

Pointers are declared similarly to primitive variables. In the code for figure 8.2.1, for example, a pointer to an **int** is declared on line 9 by specifying the type **int***. A pointer to a **double** variable would be declared with the type **double*,** and a pointer to a **char** would be declared as **char***. The star included with the declaration indicates that the variable is a pointer.

Figure 8.2.1

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    //Declare an integer
6    int myVariable = 10;
7
8    //Declare an integer pointer
9    int* myPointer;
10
11   //Assign the address of
12   //myVariable to the pointer
13   myPointer = &myVariable;
14
15   //The pointer can now be
16   //dereferenced to get the
17   //content of what it points to.
18   //To dereference a pointer, use the
19   //star in front of the pointer.
20   cout << "myPointer points to " << *myPointer << endl;
21
22   //On the other hand, if the star is not used
23   //before the pointer name, the address of the
24   //variable that myPointer points to is displayed.
25   cout << "myPointer contains " << myPointer << endl;
26 }
```

In order to obtain the address of a variable, the **address of operator** ('&') is used. Pointers are initialized with the address of a variable, as shown on line 13 of figure 8.2.1. This line stores the address of the variable **myVariable** into the pointer **myPointer**. Once this assignment has taken place, the address within the pointer can be used to manipulate the variable that is points. By **dereferencing** the pointer on line 20, the content of the variable is accessed. In this code example, the variable **myVariable** and the dereferenced pointer ***myPointe**r are acting as aliases to the same memory location. Because of this, the dereferenced pointer may be assigned a value to change the content of the variable. The code in figure 8.2.2 shown declares and initializes the pointer on line 12, and then proceeds to assign the value 99 to the dereferenced pointer on line 16. By doing this, the value stored in **myInteger** changes to 99.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //We will be using the address of this variable
6     int myInteger = 10;
7
8     //myIntegerPointer is an integer pointer, so it can hold the address
9     //of an integer.  By placing the '&' before myInteger,
10    //we are asking for the address of myInteger, NOT the value
11    //that myInteger holds.
12    int *myIntegerPointer = &myInteger;
13
14    //We can alter the value stored in myInteger by dereferencing and
15    //assigning to the pointer.
16    *myIntegerPointer = 99;
17
18    //Now myInteger will be 99, since we changed its value
19    //through myIntegerPointer
20    cout << "Now myInteger is " << myInteger << endl;
21
22    return 0;
23  }
```

Figure 8.2.2

The C++ language also provides **reference variables,** which act similarly to pointers, but with less notation. The program in figure 8.2.3 is a re-write of the code in the last example to use reference variables.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int myInteger = 10;
6
7     //This declares a reference variable and initializes it
8     //with the myInteger variable.  This is similar to
9     //declaring a pointer and assigning it the address of a
10    //variable.
11    int &referenceVariable = myInteger;
12
13    //The reference variable can be /referenced to change
14    //the variable that it references.
15    referenceVariable = 99;
```

Figure 8.2.3

```
16    //Now myInteger will be 99, since the value was
17    //changed through referenceVariable
18    cout << "Now myInteger is " << myInteger << endl;
19
20    return 0;
21  }
```

Figure 8.2.3 (cont.)

## Section 8.3: Pointers and Arrays

Pointers can hold the address of more than just a primitive variable. Arrays can also be pointed to, and their elements accessed, through a pointer. When this occurs, there are two types of notations that can be used to access a specific element in an array from a pointer. The first method is **array notation**, where a pointer is accessed using brackets and an index as if it were an array. The code in figure 8.3.1 illustrates array notation.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int myArray[] = {1,2,3};
6
7     //Declare an initialize a pointer to the array
8     int *pointerTomyArray = myArray;
9
10    //The elements in the array can be accessed
11    //through the pointer using array notation
12    cout << pointerTomyArray[0] << endl;
13    cout << pointerTomyArray[1] << endl;
14    cout << pointerTomyArray[2] << endl;
15
16    return 0;
17  }
```

Figure 8.3.1

In addition, both pointers and arrays can be accessed using **pointer notation**, which is an alternative to array notation. With pointer notation, the index is added to the pointer, and the entire expression is dereferenced. Figure 8.3.2 shows how this type of notation is expressed by displaying the 2nd element in the array through both the array name and the pointer to the array.

Pointer and array notation perform the same operation and are interchangeable. Pointer notation is useful for understanding that an element within an array is accessed by adding an index to the address of the

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int myArray[] = {1,2,3};
6     int *myPointer = myArray;
7
8     //Pointer notation using an array
9     cout << *(myArray + 1) << endl;
10
11    //Pointer notation using a pointer
12    cout << *(myPointer + 1) << endl;
13
14    return 0;
15  }
```

Figure 8.3.2

array. Array notation is more commonly used since it is easier to type, especially with multidimensional arrays.

These two notations show how how pointers and arrays can be accessed using the same notation. Though very similar, the distinction between pointers and arrays is that pointers can be change to point to something else, whereas an array cannot.

Figure 8.3.3 creates two arrays and initializes a pointer to point to **myArray** on line 10. After this assignment, the pointer is then changed to point to **myArray2**. This is a valid operation in C++ since a pointer can be assigned a different array to point to.

On line 19, however, the array **myArray2** is being assigned to **myArray**. This will cause the program to not compile since since arrays cannot be assigned to.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    int myArray[] = {1,2};
6    int myArray2[] = {3,4};
7
8    //Declare and intialize a pointer to
9    //the first array
10   int *myArrayPtr = myArray;
11
12   //The pointer can now be changed to
13   //point to the second array.
14   myArrayPtr = myArray2;
15
16   //However, an array cannot be assigned to
17   //a different array.  This line will cause the
18   //program to fail when compiling if uncommented
19   myArray = myArray2;
20
21   return 0;
22 }
```

Figure 8.3.3

## Section 8.4: Pass by Reference

When functions were presented in a previous chapter, variables were passed in as arguments. The function would then be able to use the values of the variables in the function body and return a result. By doing this, the value of the variable, and not the variable itself, was passed into the function. This is called **pass by value** since only the value is passed to the function. As a result, the function is not able to modify the variable passed as an argument. Figure 8.4.1, for example, attempts to change **myVariable**, which was passed as an argument, by assigning to the parameter. This will not work.

```
1  #include <iostream>
2  using namespace std;
3
4  void myFunction(int parameter){
5    //This doesn't change myVariable in main
6    parameter = 0;
7  }
8
9  int main(){
10   int myVariable = 99;
11   myFunction(myVariable);
12
13   cout << "myVariable is still " << myVariable << endl;
14
15   return 0;
16 }
```

Figure 8.4.1

```
1  #include <iostream>
2  using namespace std;
3
4  //Notice that we are now accepting a
5  //pointer to an integer, not just an integer
6  void doubleNumber(int *number){
7    //We dereference the pointer to get
8    //at the value of the variable that is
9    //being pointed to.
10   *number *= 2;
11 }
12
13 int main(){
14   int myNumber = 3;
15
16   //The address of myNumber is passed
17   //in now, instead of just its value
18   doubleNumber(&myNumber);
19
20   cout << myNumber << endl;
21
22   return 0;
23 }
```

Figure 8.4.2

If a pointer to **myVariable** were passed into the function, however, the value could be modified by the function. The code in figure 8.4.2 defines a function called **doubleNumber** that accepts the address of an integer. When this function is called on line 18, the address of the local variable **myNumber** is passed as an argument. Since the address, and not the value, of the variable was passed to the function, its content can be changed. The function takes the address of the number and dereferences it to assign a new value on line 10. Because of this, the number displayed by line 20 will be 6, since the value was changed. When passing variables to a function this way is called **pass by reference**.

Pass by reference may also be accomplished with a **reference variable.** Figure 8.4.3 defines the **doubleNumber** function from the previous example using reference variables instead of pointers.

The function **doubleNumber** uses a reference variable by placing the ampersand in front of the parameter name. Because of this, variables can be directly passed into the function without needing to use the address of operator. Line 18, for example, passes in **myNumber** as if it were being passed by value. But, since the parameter for **doubleNumber** is a reference variable, the value of **myNumber** in the main function can be changed by the function.

```
1  #include <iostream>
2  using namespace std;
3
4  //By placing the ampersand in front of
5  //the parameter, this is now a reference variable
6  void doubleNumber(int &number){
7    //Reference variables can be treated just
8    //like an ordinary variable/
9    number *= 2;
10 }
11
12 int main(){
13   int myNumber = 3;
14
15   //The variable can be directly passed in
16   //without using the address of operator,
17   //and the function will be able to modify its value
18   doubleNumber(myNumber);
19   cout << myNumber << endl;
20   return 0;
21 }
```

Figure 8.4.3

## Section 8.5: The Heap

In C++, there are two types of variables that can be used in a program. The first are **statically allocated** variables which are allocated on the stack. These types of variables have been used in all of the examples up to this point. A major disadvantage of static variables is their fixed size that cannot change as the program runs. This was the limitation encountered in the opening of the chapter, where the statically allocated array was not able to handle an unknown number of values that could be entered by the user.

In contrast, **dynamically allocated** variables are allocated on the heap and can change in size as the program runs. The **new** operator in C++ allocates memory on the heap and returns its address. The returned address is then stored into a pointer and dereferenced to access the memory allocated from the heap. Once the dynamic memory is no longer used, it must be deleted.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Declare a pointer that points to
6     //a piece of dynamically allocated
7     //integer array that can hold three
8     //integers.
9     int *myMemory = new int[3];
10
11    myMemory[0] = 0;
12    myMemory[1] = 1;
13    myMemory[2] = 2;
14
15    cout << myMemory[0] << endl;
16    cout << myMemory[1] << endl;
17    cout << myMemory[2] << endl;
18
19    //When the allocated memory is no longer
20    //used, it must be deleted. Without this,
21    //the memory in the heap would eventually
22    //become depleted.
23    delete [] myMemory;
24
25    return 0;
26  }
```

Figure 8.5.1

The code in figure 8.5.1 uses the **new** operator on line 9 to allocate enough memory to hold an **int** array with three elements. The memory is then accessed through the pointer to access and display the elements. Line 23 then deletes the allocated memory, which allows it to be used for future requests with the **new** operator.

Previously, a fixed array was declared to store input from the user. Using dynamic memory, an integer array can be dynamically allocated to handle input from the user. The program in figure 8.5.2 prompts the user for how many numbers will be entered, and then dynamically allocates an **int** array large enough to handle the requirement.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Ask the user how many numbers will be entered
6     int numIntegers;
7     cout << "How many integers will you enter? ";
8     cin >> numIntegers;
9
10    //Allocate a piece of memory large enough to
11    //hold everything the user enters.  Notice we
12    //are allocating an integer array.
13    int *myIntegers = new int[numIntegers];
```

Figure 8.5.2

```
14
15    //Read in the numbers one by one
16    for(int i = 0; i < numIntegers; i++){
17      cin >> myIntegers[i];
18    }
19
20    //Print out the numbers the user entered
21    for(int i = 0; i < numIntegers; i++){
22      cout << "You entered " << myIntegers[i] << endl;
23    }
24
25    //Free the memory allocated previously allocated.
26    //This is important.
27    delete [] myIntegers;
28
29    return 0;
30  }
```

Figure 8.5.2

## Section 8.6: Dynamic Memory and Arrays: Dynamic Arrays

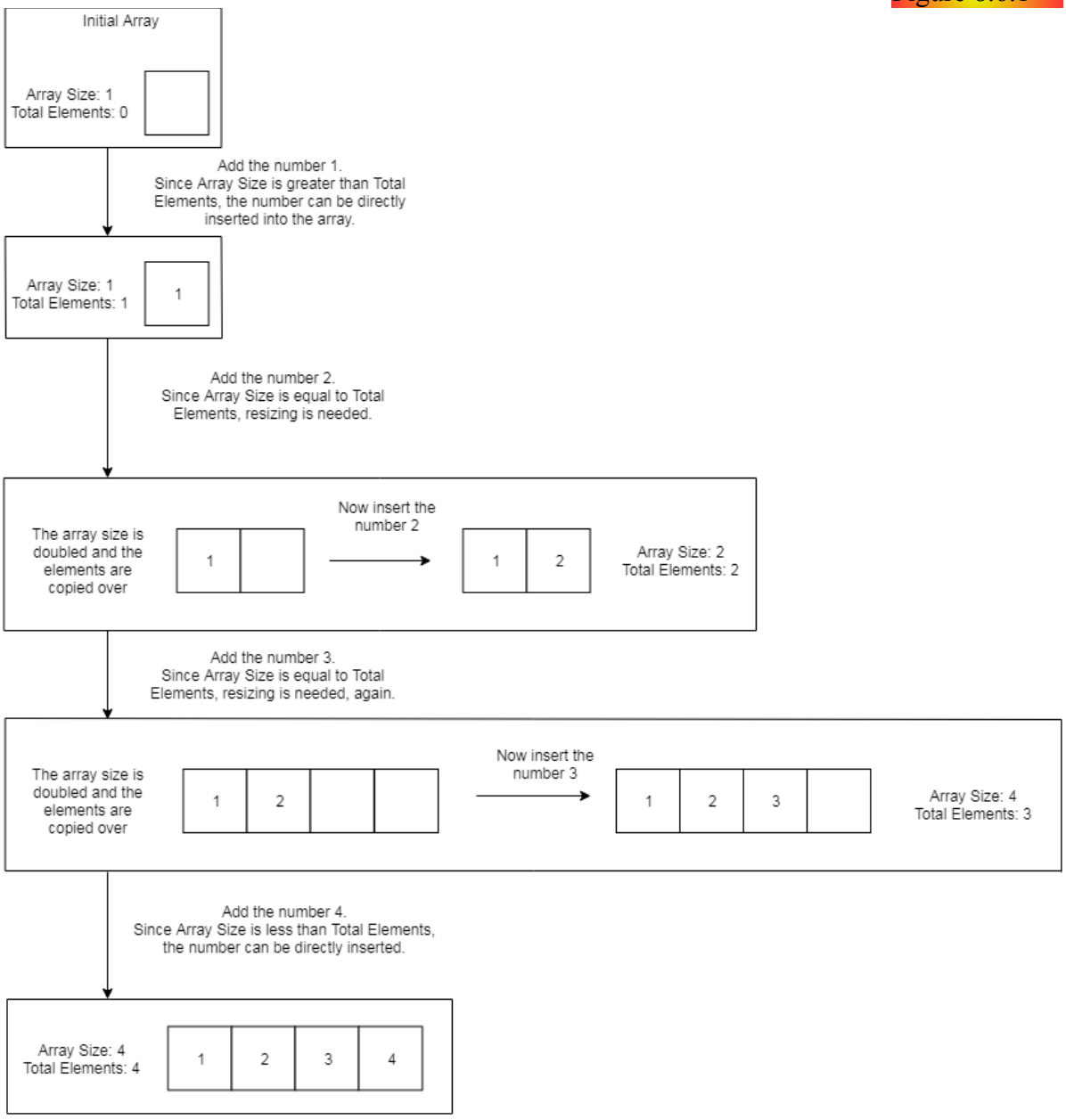
By combining dynamic memory with arrays, an array can be created which can change its size as the program runs. This would allow our programs to continually read in and store data without using a fixed size array. This type of array, called a **dynamic array**, removes the limitation of a fixed array. Dynamic arrays work by resizing the underlying array as it fills up with new values. This is similar to how a child would hunt for eggs during Easter. Initially, the smallest basket is used to hold eggs until it fills up. Once the small basket is full, the medium sized basket is grabbed, the eggs from the smallest basket are transferred over, and more eggs are gathered. Once the medium sized basket fills up, the largest basket is grabbed, the eggs from the medium sized basket are transferred over, and the process continues. While the baskets never change size, switching to a larger basket from a smaller basket allows the capacity to dynamically change.

In exactly the same way, a dynamic array is initialized by allocating a small array that can hold one element. When the array has been filled, a larger array is allocated, the values are copied over, and more values are inserted. Once the arrays fills again, an even larger array is allocated, the values are copied over, and more are inserted.

The illustration in figure 8.6.1 shows how this process works with an array. Initially, an array that has a size of 1 is dynamically allocated. To manage the dynamic array, both the capacity of the array (the Array Size) and the number of elements (the Total Elements) for the array are tracked. Initially, the array is given a size of 1 and a total number of elements of 0. When the number 1 is inserted into the array, the total number of elements is incremented. Even though the array is full, it will not be resized unless more elements are added that would

overflow its capacity. When the number 2 is inserted, the code recognizes that the array is full since Array Size is equal to Total Elements. A new array is allocated that is twice the size of the old array, and the numbers are copied over. The new Array Size is 2, and the number 2 is then inserted. When the number 3 is inserted, the code recognizes that the array is full again because Array Size is equal to Total Elements. The process of allocating a larger array, copying over the values, and inserting the new element is repeated. When the number 4 is inserted, the array has sufficient capacity to hold the new value. In this case, the number is placed into the array and no resizing occurs.

Figure 8.6.1




The code in figure 8.6.2 implements a dynamic array based on this process.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    //Allocate initial array of size 1
6    int *array = new int[1];
7    int arraySize = 1;
8    int totalElements = 0;
9
10   //Add the value of this variable to the dynamic array
11   int itemToAdd1 = 1;
12
13   //If the next free index is within the array, place the item
14   if( totalElements < arraySize ){
15     array[totalElements] = itemToAdd1;
16     totalElements += 1;
17   }else{
18     //New element cannot fit into the array, so we need to reallocate
19
20     //Temporarily point to the old array
21     int *oldArray = array;
22
23     //The size of the new array will be double the old size
24     int newSize = arraySize * 2;
25
26     //Allocate the new array with the new size
27     array = new int[newSize];
28
29     //Now copy over the elements from the old array
30     for(int i = 0; i < arraySize; i++){
31       array[i] = oldArray[i];
32     }
33
34     //Update the array size
35     arraySize = newSize;
36
37     //Delete the old array to conserve memory
38     delete [] oldArray;
39
40     //Finally, we can add the new element
41     array[totalElements] = itemToAdd1;
42     totalElements++;
43   }
44 }
```

Figure 8.6.2

In order to insert more elements into the array, the code from the last example can be duplicated as shown in figure 8.6.3.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     //Allocate initial array with a size of 1
6     int *array = new int[1];
7     int arraySize = 1;
8     int totalElements = 0;
9
10    //This value will be added to the dynamic array
11    int itemToAdd1 = 1;
12
13    //If there is free space, insert the element
14    if( totalElements < arraySize ){
15      array[totalElements] = itemToAdd1;
16      totalElements += 1;
17    }else{
18      //The array does not have enough space, so we need to reallocate
19
20      //Temporarily point to the old array
21      int *oldArray = array;
22
23      //The size of the new array will be double the old size
24      int newSize = arraySize * 2;
25
26      //Allocate the new array with the new size
27      array = new int[newSize];
28
29      //Now copy over the elements from the old array
30      for(int i = 0; i < arraySize; i++){
31        array[i] = oldArray[i];
32      }
33
34      //Update the array size
35      arraySize = newSize;
36
37      //Delete the old array to conserve memory
38      delete [] oldArray;
39
40      //Finally, the new element can be inserted
41      array[totalElements] = itemToAdd1;
42      totalElements++;
43    }
```

Figure 8.6.3

```
44    //Now insert the second element
45    int itemToAdd2 = 2;
46    //If there is space available, insert the item
47    if( totalElements < arraySize ){
48      array[totalElements] = itemToAdd1;
49      totalElements += 1;
50    }else{
51      //The array does not have enough space, so we need to reallocate
52
53      //Temporarily point to the old array
54      int *oldArray = array;
55
56      //The size of the new array will be double the old size
57      int newSize = arraySize * 2;
58
59      //Allocate the new array with the new size
60      array = new int[newSize];
61
62      //Now copy over the elements from the old array
63      for(int i = 0; i < arraySize; i++){
64        array[i] = oldArray[i];
65      }
66
67      //Update the array size
68      arraySize = newSize;
69
70      //Delete the old array to conserve memory
71      delete [] oldArray;
72
73      //Finally, insert the new element
74      array[totalElements] = itemToAdd1;
75      totalElements++;
76    }
77
78    return 0;
79  }
```

Figure 8.6.3 (cont.)

In the interest of space, the insertion for the third element has been left out. By now, you should recognize that duplicating code in this manner is bad practice and that a function should be used. However, for a function to handle a dynamic array, the variables **array**, **arraySize**, and **totalElements** would need to be changed by the function. This would require the use of pointers to each of these variables. In addition, the number being inserted to the array would need to be added as a parameter to the function. The function prototype for this function is shown in figure 8.6.4.

```
1    void addToDynamicArray(int *array, int *arraySize, int *totalElements, int element);
```

Figure 8.6.4

Both **arraySize** and **totalElements** need to be passed as pointers since the function must modify their values. To an experienced C++ programmer, a function that takes four parameters, three of which are pointers, looks wrong. Until this chapter, functions have been utilized to solve code duplication. But, due to the an increasing number of variables that are managed, functions are becoming cumbersome to use. In order to solve this problem, both the data and the code for the dynamic array should be brought together. Instead of a function operating on multiple pieces of information, it would be far better for both code and data to be unified and isolated into its own unit. Objects do exactly this, and they will be explored in the next chapter.

# Chapter 9: Objects

Chapter Layout



## Section 9.1: Why Objects?

A program fundamentally consists of data and code. Up until this chapter, both of these have been separate, with data being represented by variables and code being represented by functions and control structures. As programs become larger, it becomes more difficult to keep the related code and data together and to isolate various parts of a program. It is far easier for humans to understand discrete systems working together rather than a large amount of code intertwined in functionality. The goal of this chapter is to utilize objects to unify code and data in addition to isolating various components of our programs into distinct and logical units. Objects are a vast topic, and semester-long courses and books have been dedicated to learning just this aspect of programming. This chapter will cover the basics of objects, their purpose, and how to use them in your programs.

## Section 9.2: Classes and Objects

Objects make programs easier to understand by providing greater fidelity to what humans understand. Objects are meant to relate to physical objects which have certain properties, represented by data, and specific functions, represented by code. A car, for example, has properties such as color, model, and year and certain functionality such as accelerate, brake, and refuel. Turning a car into an object involves replacing the properties with variables and the functionality with functions.

In order to create an object, it is instantiated from a class. A class acts as a blueprint for the object, much as a specific, physical car comes from the specifications for the model. A class defines how the object is structured, including the data that it holds as well as the functionality it defines. Classes are types that can be used to declare variables, just as the primitive types **int** or **double** are used to declare variables within a program. In essence, objects allow programmers to create new types to be used in a program.

By using classes, new types can be defined within the C++ language and be used by the program. Instead of being constrained to representing data as single variables, a class can be used to unify many related variables and code into a single package. This process is called **encapsulation** and allows programmers to abstract the underlying code.

## Section 9.3: A Basic Object

In order to understand the relationship between classes and objects, consider a program which uses a counter to record the number of times an event occurs. To write this program, a variable could be used to hold the current count and the increment operator could be used to increment the variable. Figure 9.3.1 shows how this code would be structured.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5     int counter = 0;
6
7     //Increment the counter
8     counter++;
9
10    //Increment again
11    counter++;
12
13    cout << "Counter is at " << counter << endl;
14
15    return 0;
16  }
```

Figure 9.3.1

If multiple counters were used, multiple counter variables would be required. In addition, the count counter variable could mistakenly be reassigned by other code. By using a class to define the counter object, the code and data can be brought together in addition to providing access specifiers for data.

The class declaration in figure 9.3.2 combines the data, which is the **counter** variable, along with functions to operate on the data.

```
1   #include <iostream>
2   using namespace std;
3
4   //Create a class called Counter
5   //This acts as its own type in our program
6   class Counter{
7     //This is the private access specifier.
8     //Everything in this section can only be accessed
9     //by code within the object.
10    private:
11      int counter;
```

Figure 9.3.2

Just as a function is used to encapsulate code, a class is used to encapsulate code and data. When used in a class, variables are called **member variables** and functions are called **member functions** or **methods**. More generally, both variables and functions within a class are referred to as **members**.

A class lays out how the object will be structured and consolidates code and data into a single package. In addition, classes provide **access specifiers** for members. The code in figure 9.3.2 illustrates this by placing the **counter** variable beneath the private access specifier on line 10. Because of this, the variable can only be accessed by the methods that are within the class. In addition, class members can be listed under the public access specifier to allow code outside of the object to access them. By default, all of the members within a class are private to disallow modification by external code.

```
12    //This is the public access specifier.
13    //Anything within this section can be accessed
14    //outside of the class.
15    public:
16      //The constructor function for this class.
17      Counter(){
18        counter = 0;
19      }
20
21    //Increments the counter variable
22    void increment(){
23      counter++;
24    }
25
26    //Returns the value of the counter
27    int getValue(){
28      return counter;
29    }
30  };
31
32  int main(){
33    //Instantiate an object of type Counter
34    //This is exactly like declaring a variable
35    Counter myCounter;
36
37    //Call the increment method on the object by using
38    //the dot operator followed by the method name.
39    myCounter.increment();
40    myCounter.increment();
41
42    //Call the getValue method by using the dot
43    //operator followed by the method name
44    cout << myCounter.getValue() << endl;
45
46    return 0;
47  }
```

Figure 9.3.2 (cont.)

Once a class has been declared, an object can be **instantiated** and used in the program. Instantiating an object is exactly the same as declaring a primitive variable, as can be seen on line 35 where an object of type **Counter** is declared.

Inside of the object beginning on line 17 is a special function called a **constructor**. A constructor has the same name as the class and has no return type. The job of a constructor is to initializes the object when it is initially declared. In this example, the counter should be initialized to 0 when first instantiated, which is handled by the constructor.

After the object has been instantiated, the method increment is called on lines 39 and 40. Since the code and data are part of the same object, the method does not need to accept arguments.

## Section 9.4: Pointers to Objects

A pointer may be used reference an object, just as if it were referencing a primitive variable. In order to access the members of an object, the pointer must first be dereferenced and then accessed using the dot operator. The code in figure 9.4.1 accomplishes this on line 42. In addition, accessing an object through a pointer can be abbreviated with the arrow operator, as can be seen on line 50. Both forms of referencing perform the same action, but the arrow operator is often preferred, especially when dereferencing multiple levels of pointers.

```
1  #include <iostream>
2  using namespace std;
3
4  //Create a class called Counter.
5  //This acts as its own type in our program
6  class Counter{
7    //This is the private access specifier.
8    //Everything in this section can only be accessed
9    //by code within the object.
10   private:
11     int counter;
12
13   //This is the public access specifier.
14   //Anything within this section can be accessed
15   //outside of the class.
16   public:
17     //The constructor function for this class.
18     Counter(){
19       counter = 0;
20     }
21   //Increments the counter variable
22   void increment(){
23     counter++;
24   }
```

Figure 9.4.1

```
25    //Returns the value of the counter
26    int getValue(){
27       return counter;
28    }
29 };
30
31 int main(){
32    Counter myCounter;
33    Counter* counterPointer = &myCounter;
34
35    //Increment the counter through the pointer
36    //First, the pointer is dereferenced by applying the
37    //star operator to counterPointer.  This part needs to
38    //be in parenthesis since the dot operator has higher
39    //precedence than the star operator.  Once the pointer
40    //has been dereferenced, then the dot operator is
41    //used to envoke the method.
42    (*counterPointer).increment();
43
44    //The previous statement can be simplified by using
45    //the arrow operator, which is exactly the same as
46    //using the star operator followed by the dot
47    //operator as shown above.
48    counterPointer->increment();
49
50    cout << counterPointer->getValue() << endl;
51
52    return 0;
53 }
```

Figure 9.4.1 (cont.)

## Section 9.5: Dynamic Array Class

The dynamic array implementation from a previous chapter was difficult to implement due to the number of variables that needed to be managed. By utilizing objects to glue together the data and code associated with a dynamic array, the methods of the class can directly access the private member variables without needing to accept arguments. The following code re-writes the dynamic array code to use a class instead of functions and variables.

```
1  #include <iostream>
2  using namespace std;
3
4  //Create a class for a dynamic array
5  class DynamicArray{
6    //These variables should not be accessed
7    //by anything outside of the object.
8    private:
9      //A pointer to a set of integers
10     int *elements;
11     //The amount of memory allocated for the elements.
12     int elementsSize;
13     //The next free space to place an element
14     int numberItems;
15
16   public:
17     //Constructor for a new DynamicArray
18     DynamicArray(int startSize);
19
20     //Returns the number of elements in the array,
21     //which is numberItems
22     int getSize();
23
24     //Adds a new integer to the end of the dynamic array
25     void append(int newNumber);
26
27     //Returns the integer at the given index
28     int getAtIndex(int index);
29
30     //Assigns the value to the given index in elements
31     void setAtIndex(int index, int val);
32 };
33
34 //The constructor for the array handles initial setup.
35 DynamicArray::DynamicArray(int startSize){
36   //First free index is at index 0 to start
37   numberItems = 0;
38
39   //Set the start size
40   elementsSize = startSize;
41
42   //Allocate an integer array to hold elements
43   elements = new int[startSize];
44 }
```

Figure 9.5.1

```
45 //Returns the number of elements held in the array
46 int DynamicArray::getSize(){
47    return numberItems;
48 }
49
50 //Appends the new number to the end of the dynamic
51 //array and resizes if needed
52 void DynamicArray::append(int newNumber){
53    //If there is enough space free...
54    if( numberItems < elementsSize){
55       //...append to the end of the array and
56       //set the next index as free
57       elements[numberItems] = newNumber;
58       numberItems++;
59    }else{
60       //We have run out of space, so the array needs
61       //to be re-allocated.
62
63       //Save a temporary pointer to the old array
64       int *temp = elements;
65
66       //Allocate twice the size for the new array
67       elements = new int[elementsSize*2];
68
69       //Copy over elements from the old array to the new array
70       for(int i = 0; i < elementsSize; i++){
71          elements[i] = temp[i];
72       }
73
74       //The size of the new array is twice the old size,
75       //so double elementsSize and free the old array
76       elementsSize *= 2;
77       delete [] temp;
78
79       //Now the new element can be placed
80       elements[numberItems] = newNumber;
81       numberItems++;
82    }
83 }
```

Figure 9.5.1 (cont.)

```
84  //Returns the integer at the given index, else
85  //0 if the index is out of range
86  int DynamicArray::getAtIndex(int index){
87    if( index < numberItems ){
88      return elements[index];
89    }else{
90      return 0;
91    }
92  }
93
94  //Sets the given element in the array if the index
95  //is within range.
96  void DynamicArray::setAtIndex(int index, int val){
97    if( index < numberItems ){
98       elements[index] = val;
99    }
100 }
101
102 int main() {
103   //Create a dynamic array with an initial capacity
104   //of one element. This line invokes the constructor
105   //which initializes the dynamic array.
106   DynamicArray a(1);
107
108   //Append a series of elements to the array
109   a.append(10);
110   a.append(11);
111   a.append(12);
112   a.append(13);
113   a.append(14);
114
115   //Use the dynamic array's methods to print out
116   //its content
117   for(int i = 0; i < a.getSize(); i++){
118     cout << "Element #" << i << " is " << a.getAtIndex(i) << endl;
119   }
120
121   return 0;
123 }
```

Figure 9.5.1 (cont.)

In this example, the methods are defined outside of the class. To define the method outside of the class, the prefix "DynamicArray::" is included in front of the method name to indicate that the function is part of the class DynamicArray. In the future, separating the class declaration from the methods definitions will be useful for splitting the code into multiple files.

By turning this code into a class, the variables that are used to manage the dynamic array can be directly accessed by the methods for the class. This simplifies the code and is easier to use as compared to functions. In addition, using a class allows the private access specifier to restrict access to the variables **elements**, **elementsSize**, and **numberItems.** Because these variables need to be consistent for the dynamic array to work correctly, restring access to only the methods within the class reduces the number of problems that could occur.

## Section 9.6: The vector Class

The C++ language comes with several libraries which contain many different classes. The vector class is particularly useful since it acts as a dynamic array that can hold any type of data. The code in figure 9.6.1 declares a vector on line 9 by specifying the type that the vector will hold in angle brackets. Values can be inserted to the end of the vector by using the push_back method for the vector object. In addition, vectors can be accessed just like arrays by using square bracket notation.

```
1   #include <iostream>
2   //This lets us use the vector class
3   #include <vector>
4   using namespace std;
5
6   int main(){
7     //When declaring a vector, we put the type that it holds
8     //With angle brackets, like "<int>"
9     vector<int> myVector;
10
11    //We can add to a vector using the push_back method
12    myVector.push_back(1);
13    myVector.push_back(2);
14    myVector.push_back(3);
15
16    //A vector can be processed in a loop just like an array.
17    //Unlike arrays, we can determine the size of a vector
18    //by calling the method size on the object.
19    for(int i = 0; i < myVector.size(); i++){
20      //Vectors can be accessed using an indexing
21      cout << myVector[i] << endl;
22    }
23
24    return 0;
25  }
```

Figure 9.6.1

The Standard Template Library comes with the C++ language and provides many useful classes that are commonly used in software development. By using the STL, programmers are able to quickly use common data structures and algorithms without needing to implement and debug the code. Using libraries makes programmers more efficient since the code has already been written and tested. While it is not necessary to understand the implementation in order to use the object, it is still import to understand how the code functions underneath.

# Chapter 10: File Input and Output

Chapter Layout



## Section 10.1: Why Are Files Needed?

In all of the previous examples, the data being processed has been stored in variables and is lost once the program ends. To be more useful, programs should be able to persistently store data and be able to retrieve it at a later time. Files serve this purpose by providing a place to store information that is retained once the program ends.

Whereas variables can be directly accessed and used by a program, files are managed by the operating system and cannot be directly accessed. To use files, a C++ program will use certain methods that ask the operating system to perform an action on a file. Since all file operations must go through the operating system, permissions may be assigned to users so that the operating system only allows certain users to access specific files. This ensures that only authorized users may have access to a file, while unauthorized users may not.

## Section 10.2: Basic File Operations

Just as variables must be declared in order to be used in a program, a file must first be **opened** in order to have operations performed on it. When opening a file, there are different options that can be provided to change how the file will be used in the program. These options control aspects of the file such the ability to read and write to the file, and how the content is interpreted. Just as dynamically allocated memory must be deleted when it is no longer used, file must be **closed** once they are no longer used by the program.

When a file has been opened, data may be written using the stream insertion operator and read using the stream extraction operator. This works the same way as displaying text to the screen using **cout** and reading user input with **cin**. Unlike displaying to the screen, where every character must be sequentially displayed, and unlike user input, where every key entered must be sequentially read by the program, data may be read and written from any part of a file. This is

called a **random-access file**, and works by setting the current position within a file. This is similar to how a cursor functions in a word processor. The cursor may be moved around, and data may be read and written at the cursor position.

## Section 10.3: Basic File Input and Output

The program in figure 10.3.1 demonstrates the basic operations of opening, writing, and closing a file.

```
1   #include <iostream>
2   //The file stream library, used for file input and output
3   #include <fstream>
4   using namespace std;
5
6   int main(){
7     //An Output file stream object, used for writing to a file
8     ofstream myFile;
9
10    //ofstream is an object that uses the open method
11    //to open the given file
12    myFile.open("test.txt");
13
14    //Write to the file using the stream insertion operator
15    myFile << "Hello, World!";
16
17    //Call the ".close()" method on the file.
18    myFile.close();
19
20    return 0;
21  }
```

Figure 10.3.1

Line 8 of figure 10.3.1 declares a variable named **myFile** with the type **ofstream**. The **ofstream** type comes from the **fstream** library and is used for outputting to a file. To write data to a file, the stream insertion operator can be used as demonstrated on line 15. Writing to a file works the same as writing to **cout**. Since the file is no longer used by the program, the close method is called on line 18. After this program has executed, the file *test.txt* will be created and will contain the text "Hello, World!". The content of this file can be read back into the program at a later time, as shown in figure 10.3.2.

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     //An input file stream object, used for reading from a file
7     ifstream myFile;
8     myFile.open("test.txt");
9
10    //Store the content of the file into a C++ string
11    string fileContent;
12    myFile >> fileContent;
13
14    cout << fileContent;
15
16    myFile.close();
17
18    return 0;
19  }
```

Figure 10.3.2

To input data from a file, the **ifstream** object is used. The content can then be read into a **string** variable by using the stream extraction operator, just as data is read from the user using the **cin** object. The file is then closed on line 16. However, when the program is executed, the text "Hello," will be displayed and not "Hello, World!" as shown in the output in figure 10.3.3.

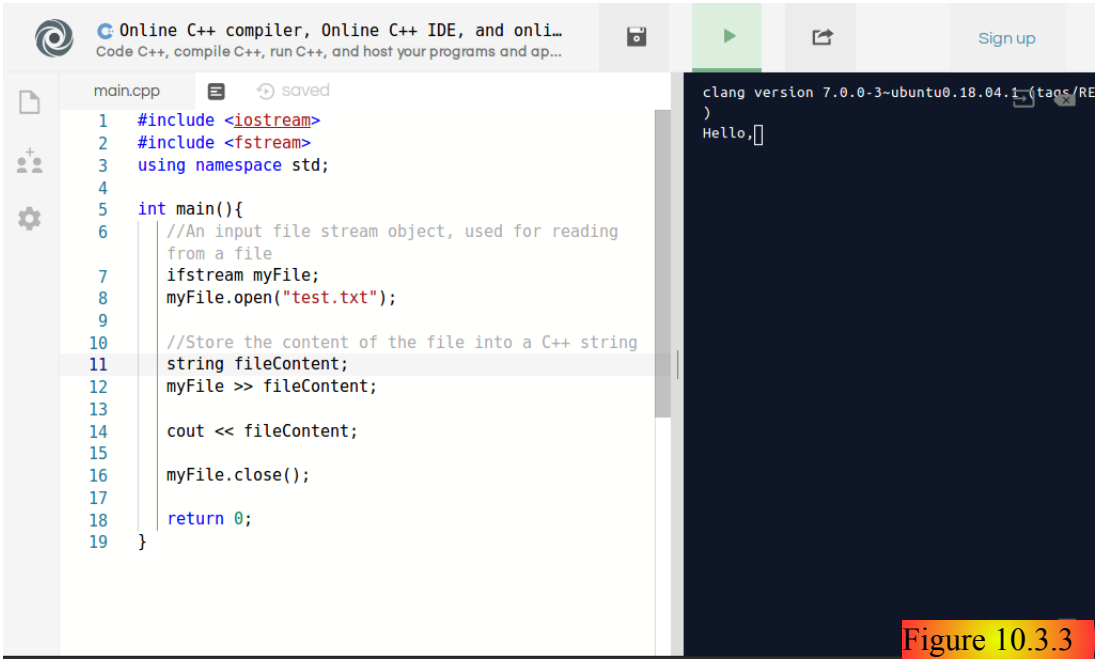



Figure 10.3.3

This occurs because the stream extraction operator delimits on spaces. Just as the end of a chapter separates sections in a book for a human, spaces in a string separates the string into pieces for the stream extraction operator. When a space is encountered, nothing more is read unless the stream extraction operator is used again on the data source. To read the entire line in the file, the stream extraction operator could be used multiple times.

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     //An input File stream object, used for reading from a file
7     ifstream myFile;
8     myFile.open("test.txt");
9
10    //Store the first part of the text into a C++ string
11    string fileContent;
12    myFile >> fileContent;
13
14    cout << fileContent;
15
16    //Read the next part of the file
17    myFile >> fileContent;
18
19    cout << fileContent;
20
21    myFile.close();
22
23    return 0;
24  }
```

Figure 10.3.4

This method works, but is clunky since the exact number of spaces in the file may not be known. Instead of using the stream extraction operator, the **getline** function can be used to retrieve an entire line from the file. Whereas the stream extraction operator delimits on spaces, the **getline** function, by default, delimits on the newline character ('\n') which marks the end of the line. The code in figure 10.3.5 uses **getline** to read the entire line of the *test.txt* file into the program.

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     //An input file stream object, used for reading from a file
7     ifstream myFile;
8     myFile.open("test.txt");
9
10    //Store the content of the file into a C++ string
11    string fileContent;
12    getline(myFile, fileContent);
13
14    //This will now correctly display "Hello, World!"
15    cout << fileContent;
16
17    myFile.close();
18
19    return 0;
20  }
```

Figure 10.3.5

Because **getline** retrieves an entire line at a time, the program correctly outputs the contents of the file. In some cases, a file may need to be opened for both reading and writing. Since **ofstream** can only be used to write to files and **ifstream** can only be used to read from files, the more general **fstream** object may be used instead. When opening a file with a **fstream** object, certain flags may be passed to the open method to both read

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     //An fstream object may be used to open a file
7     //for both reading and writing
8     fstream myFile;
9
10    //Open the file for both reading (ios::in) and writing (ios::out)
11    //by bitwise oring the ios::in and ios::out flags
12    myFile.open("test.txt", ios::in | ios::out);
13
14    //The file may be read from
15    string inputLine;
16    getline(myFile, inputLine);
17    cout << inputLine << endl;
18
19    //The file may also be written to
20    myFile << "This is a test" << endl;
21
22    return 0;
23  }
```

Figure 10.3.7

and write to the file. The code in figure 10.3.7 performs this by bitwise oring the ios::in and ios::out flags.

## Section 10.4: Random Access with Files

The previous examples showed how to sequentially read and write to a file. Using sequential access, data in the middle of the file could only be read by reading all preceding data. Sequential access is similar to how TV broadcasting worked, where each program was played in series and could only be viewed at a certain time. Random access, on the other hand, works like a streaming service, where any episode can be accessed on demand.

In order to randomly access data within a file, the current position within the file can be moved to the desired location. Just as text may be read and written at the cursor position in a word processor, data can be read and written to the current position in a file. Moving the position within a file involves using the **seekg** and **seekp** methods for file objects. In order to find the position within a file, the **tellg** and **tellp** can be used. For the purposes of working with files, the **seekg** and **seekp** are equivalent, as well as **tellg** and **tellp**.

Setting the file position works similarly to indexing within an array. An offset in bytes is added to either the beginning (ios::beg), the end (ios::end), or the current position (ios::cur) within the file to move to the new position.

The previous example code wrote the text "Hello, World!" to a file named *test.txt*. The following program in figure 10.4.1 uses the **seekg** method to position the cursor over the "W" within the file. Since there are seven characters from the beginning of the string "Hello, World!" to the "W", the **seekg** method will set an offset of 7 from the beginning of the file. When the cursor has been positioned, the stream extraction operator is then used to store "World!" into **fileContent**.

Figure 10.4.1

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     //Create an input file stream and open test.txt
7     ifstream myFile;
8     myFile.open("test.txt");
9
10    //The file contains "Hello, World!"
11    //Which means the "W" is 7 bytes from the beginning
12    //of the file.  Position 7 bytes from the beginning of
13    //the file byadding 7 to the beginning of the file (ios::beg)
14    myFile.seekg(7, ios::beg);
15
16    //Store the word into fileContent
17    string fileContent;
18    myFile >> fileContent;
19
20    cout << fileContent;
```

To locate the current position within a file, the **tellg** method can be used. This method returns an integer specifying the byte offset relative to the beginning of the file (ios::beg). The code in figure 10.4.2 displays the initial position within the file, moves to an offset of 5 within the file, and then displays the position again.

```
21    //The "W" is the 6th character from the end, so it has
22    //an offset of -6 from the end of the file.
23    //This is the same position as before, but expressed
24    //from the end of the file (ios::end).
25    myFile.seekg(-6, ios::end);
26
27    //The content will be the same as before
28    myFile >> fileContent;
29    cout << fileContent;
30
31    return 0;
32  }
```

Figure 10.4.1 (cont.)

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     ifstream myFile;
7     myFile.open("test.txt");
8
9     //When opening a file, the file position start at
10    //the beginning (offset 0).
11    cout << myFile.tellg();
12
13    //Now position to offset 5
14    myFile.seekg(5, ios::beg);
15
16    //The position will now be 5
17    cout << myFile.tellg();
18
19    return 0;
20  }
```

Figure 10.4.2

## Section 10.5: Processing Files with Loops

By storing information in a file, a C++ program can loop through the content and process the information. As an example, suppose a file has a series of numbers on different lines as shown in figure 10.5.1. The first number on each line specifies if the shape is a triangle (0), square (1), or circle (2) followed by the measurements of the specific shape. The goal of the program will be to calculate the area of each shape

```
1   0 3 1
2   2 9
3   1 3
4   1 6
5   0 9 8
6   2 3
7   1 9
```

shapes.txt

Figure 10.5.1

specified. For example, if the line "0 2 3" is read, then the program will calculate the area of a triangle with a base of length 2 and a height of 3. If the line is "1 7", then the program will calculate the area of a square with sides of length 7. Alternatively, if the line is "2 4", then the program will calculate the area of a circle of radius 4. The file s*hapes.txt* will contain the measurements for the program to process. Create this file in repl.it by going to the "Add File" button on the left hand side, naming the file *shapes.txt*, and inserting the content of figure 10.5.1.

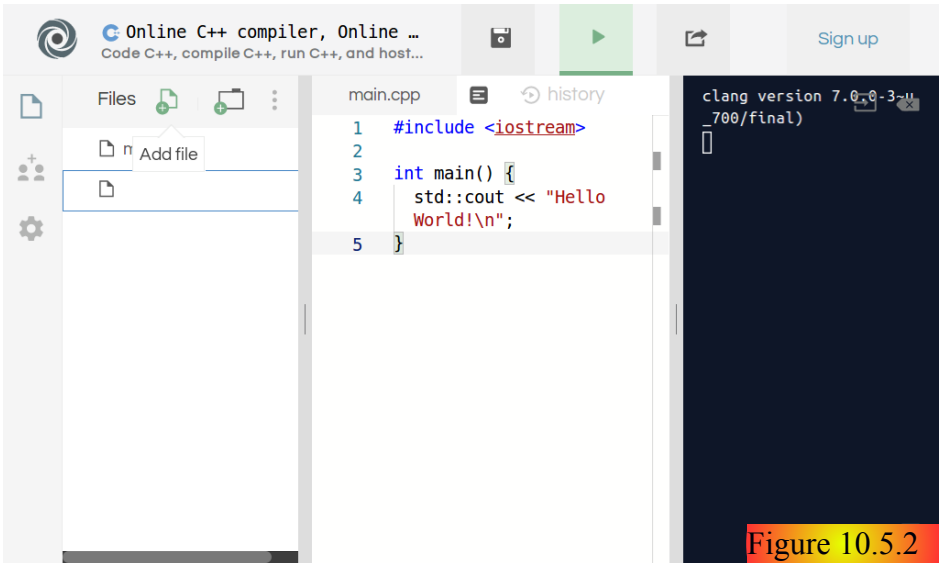


Figure 10.5.2

After the information has been entered into *shapes.txt*, switch to *main.cpp* and enter the following code.

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  //These constants will be used in the program
6  //for specifying what type of shape we are
7  //dealing with.  Each line in the file will begin
8  //with one of these values, and follow with the
9  //relevant measurements for the shape.
10 const int TYPE_TRIANGLE = 0;
11 const int TYPE_SQUARE = 1;
12 const int TYPE_CIRCLE = 2;
13 const double PI = 3.14;
14
15 int main() {
16   //Create an input stream for the input file
17   ifstream inputShapes;
18
```

Figure 10.5.3

```
19  //Open the Shapes.txt file to read in the file
20  inputShapes.open("Shapes.txt");
21
22  //This will hold the first number on the line,
23  //which should have a value equal to one
24  //of the constants declared above
25  int shapeType;
26
27
28  //Process each line in the file
29  do{
30    //Read in the first number on the line
31    //to get the type of the shape.
32    inputShapes >> shapeType;
33
34    //For triangles...
35    if( shapeType == TYPE_TRIANGLE){
36      //We need to read in two values
37      //(the height and base) to calculate
38      //the overall area
39      double base, height;
40
41      //Read in both values in a single line
42      inputShapes >> base >> height;
43
44      //Area = (1/2) * base * height
45      cout << "Area of triangle: " << .5 * base * height << endl;
46
47    //For squares...
48    }else if( shapeType == TYPE_SQUARE){
49      //We only have a single number, the
50      //length of a side
51      double length;
52      inputShapes >> length;
53
54      //Area = length * length
55      cout << "Area of square: " << length * length << endl;
56
57    //For circles...
58    }else if( shapeType == TYPE_CIRCLE){
59      //Read in the radius of the circle
60      double radius;
61      inputShapes >> radius;
62
```

Figure 10.5.3 (cont.)

```
63      //Area = PI * (r * r)
64      cout << "Area of circle " << radius * radius * PI << endl;
65
66    //Invalid shape.
67    }else{
68
69      cout << "Unrecognized shape" << endl;
70    }
71
72    //Continue to loop through the file until
73    //the end of the file is reached.
74  }while( !inputShapes.eof() );
75
76  return 0;
77 }
```

Figure 10.5.3 (cont.)

The condition on line 74 tests if the end of the file has been reached by calling the eof method, which stands for end of file. So long as there is still data in the file to be read, the result of calling inputShapes.eof() will be false. The boolean negation (the "!") is added to flip this, so that !inputShapes.eof() is true as long as there is data to read and false when the end of the file has been reached to terminate the loop

## Section 10.6: Binary Files

By default, files are opened in text mode, interprets the content of a file as ASCII characters. Files may also be opened in binary mode, which allows arbitrarily formatted data to be stored and retrieved.

By using binary mode when performing file input and output, an array can be directly written to a file and retrieved at a later time. The code in figure 10.6.1 does this by opening the file in binary mode with the ios::binary flag provided to the open method.

When the program to the right has been executed, the file *test.bin* will have been created with the content of the array. If the file is opened by clicking on it within the online interface, the numbers 5,

Figure 10.6.1

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     ofstream binaryFile;
7
8     //Open file in r/w binary mode
9     binaryFile.open("test.bin", ios::binary);
10
11    //Declare an array to write to the file
12    int myArray[] = {5, 2, 4, 1, 2, 4};
13
14    //Write the array to the binary file.  The first
15    //argument is the name of the array, and the
16
```

2, 4, 1, 2, and 4 will be interpreted as ASCII characters, even through the data was written as binary. Because these numbers are non-printable ASCII characters, the output will look strange since the numbers are being interpreted as characters.

```
17    //second is its size.  The size of this array is
18    //the size of each individual element (sizeof(int))
19    //times the number in the array (6)
20    binaryFile.write(myArray, sizeof(int) * 6);
21
22    binaryFile.close();
23
24    return 0;
25  }
```

Figure 10.6.1 (cont.)

This highlights an important aspect of dealing with data in a program. Data, by itself, has no inherent meaning and only gains a meaning by its interpretation. The number 97, for example, could mean the integer value 97, or the ASCII character 'a', or even the hexadecimal value 97. The number 97 does not change, but its meaning does depending upon how it is interpreted. This is why data read back from a binary file must be interpreted in the correct way. The program in figure 10.6.2, for example, must interpret the file contents as an array of 6 integers in order to correctly represent the data that was originally written to the file.

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main(){
6     ifstream binaryFile;
7
8     //Use the ios::binary flag to open the file in binary mode
9     binaryFile.open("test.txt", ios::binary);
10
11    //The data will be placed in this array
12    int array[6];
13
14    //Read the data back into an array of the correct size
15    //Note that we need to know the size of this array
16    binaryFile.read(array, sizeof(int) * 6);
17
18    //Print out content
19    for(int i = 0; i < 6; i++){
20      cout << array[i] << endl;
21    }
22
23    return 0;
24  }
```

Figure 10.6.2

Since the binary format for files is so flexible, new file types may be created by specifying the format of the data. A program may then be written to correctly interpret data read and written to the file that conforms with the format. This is how file formats such as pdf, docx, and mp3 files are created.

## Section 10.7: Letter Frequency in a File

By using **getline** and a loop, we have the ability to loop through every line within a file. Another loop can then be used to loop over every character in the line and count how many times each character appears. The program in figure 10.7.1 implements this in order to calculate the letter frequency within a file.

Figure 10.7.1

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main(){
6    string fileName;
7    fstream inputFile;
8
9    //Initialize each element in the array to be 0.
10   int frequencies[256];
11   for(int i = 0; i < 256; i++){
12     frequencies[i] = 0;
13   }
14
15   //Prompt user for file name
16   cout << "Enter file name: ";
17   cin >> fileName;
18
19   inputFile.open(fileName, ios::in);
20
21   //Get the first line of the file
22   string inputLine;
23   getline(inputFile, inputLine);
24
25   //Continue to grab lines from the file
26   //while we have not hit EOF
27   while( ! inputFile.eof() ){
28     //Use another loop to process each line
29     for(int i = 0; i < inputLine.length(); i++){
30       char readCharacter = inputLine[i];
31       frequencies[readCharacter] += 1;
32     }
33   getline(inputFile, inputLine);
34   }
35   //While we counted the frequencies of all ascii characters,
36   //we will only display the printable characters.
```

By using the nested loops on lines 27 and 29, the program is over to loop over every line in the file, and then, for each line, loop over each character in the line.

Lines 30 and 31 contain an indexing trick that can be used to easily keep track of the number of characters of each type. Characters are really just integers, which means that they can be used as indexes for an array. For example, if the character 'a' is read, then the corresponding ASCII value of 97 is used as an index in the array.

Line 39 contains another trick with types. The loop variable **i** is casted into a character and sent to **cout**. Since the loop variable is going from the values 32 to

126, casting turns the integer into an ASCII character. By casting the number to a character, the character is displayed on the line.

```
37    cout << "Frequencies: " << endl;
38    for(int i = 32; i < 127; i++){
39       cout << (char)i << ": " << frequencies[i] << endl;
40    }
41
42    return 0;
43 }
```

Figure 10.7.1 (cont.)

# Chapter 11: Multi-file Projects

Chapter Layout



## Section 11.1: Why Multiple Files?

Many of the programming constructs you have been using were created in order to reduce the amount of code that needs to be written when developing software. As programs become more complex, separating related code into multiple files reduces the amount of effort required to maintain and modify the program. Just as functions were used to group together related code, multiple files are used to group together related functions, objects, and data. By separating code according to purpose into different files, the program becomes modular and easier to work with.

In addition to easing the burden on programmers, separating code into different files allows libraries to be created and used. Libraries in C++, such as **iostream** and **fstream**, are files that contain code and are used by your program. All of the examples until now have used a single source code file, but professionally developed programs will almost certainly contain multiple files.

## Section 11.2: Basics of Multi-file Projects

When separating code in a project, header files and source code files are used. Each type of file has a specific purpose which dictates where code and declarations should be placed. All of the previous examples have used a single **source file** for the entire program. A source file ends with the extension **.cpp** to indicate that it contains C++ code. In a multi-file project, executable code such as functions and methods are placed into source files. Declarations such as function prototypes and class declarations are placed into **header files** that end with the **.h** extension. One important rule for multi-file projects is that executable code should never be placed in a header file.

Source and header files are commonly paired together by splitting the declarations into a header file and the associated code within a source file. Just as objects create an interface for outside code using public methods, the header file serves a similar purpose by providing an

interface for other code to use. The source code file acts like the private member functions of an object by hiding implementation details from other programmers.

The process of bringing together all of the separate files into a single executable is called **building** the project, and is automatically performed by the online interface. Other programming environments, such as an Integrated Development Environment or a command line interface, will build projects differently.

## Section 11.3: Headers and Source Files

In order to see how a multi-file project evolves from a single file project, consider the program in figure 11.3.1 that uses the function **sayHello** to display a message to the screen. In order to make the program more modular, the **sayHello** function will be moved into its own file. To do this, first create a new file in the online interface called *sayhello.cpp* and type in the code in figure 11.3.3. In addition, the function is replaced with its prototype in *main.cpp*.

```
1  #include <iostream>
2  using namespace std;
3
4  void sayHello(){
5    cout << "Hello!" << endl;
6  }
7
8  int main() {
9    sayHello();
10 }
```

Figure 11.3.1

**main.cpp**

```
1  #include <iostream>
2  using namespace std;
3
4  //Since the actual function is defined
5  //in a different file, we need the
6  //prototype so that is can be called
7  //by the main function.
8  void sayHello();
9
10 int main() {
11   sayHello();
12 }
```

Figure 11.3.2

**sayHello.cpp**

```
1  #include <iostream>
2  using namespace std;
3
4  void sayHello(){
5    cout << "Hello!" << endl;
6  }
```

Figure 11.3.3

The program will perform the same as before. Since the **sayHello** function was moved into a different file, the function prototype needs to be included in *main.cpp* so that it can be called by the main function. For this example, writing the function prototype for **sayHello** does not require much effort. However, real-world software projects will often define dozens of functions in separate files. Writing the function prototypes for each would be tedious and error-prone. To solve this, the function prototypes are placed into a header file and included in the

main program. Create the file *sayHello.h* and enter the code shown in figure 11.3.6. In addition, the *main.cpp* file is updated to include *sayHello.h* instead of directly entering the function prototypes.

**main.cpp**

```
1  //Angle brackets are used
2  //since iostream is a system library
3  #include <iostream>
4  using namespace std;
5
6  //Instead of manually entering each
7  //function prototype, we can include
8  //them from sayHello.h.  Double
9  //quotes are used here since sayHello.h
10 //is in our project and not in the
11 //system library
12 //This line...
13 #include "sayHello.h"
14 //...is replaced with the content of the
15 //sayHello.h file, which is
16 // “void sayHello();”
17
18 int main() {
19   sayHello();
20 }
```

Figure 11.3.4

**sayHello.cpp**

```
1  #include <iostream>
2  using namespace std;
3
4  void sayHello(){
5    cout << "Hello!" << endl;
6  }
```

Figure 11.3.5

**sayHello.h**

```
1  void sayHello();
```

Figure 11.3.6

By placing the function prototype into a header file and including it into *main.cpp*, the prototypes can be quickly added without needing to read through the code in *sayHello.cpp*. The diagram below shows how the main program, the source file, and the header file are related to one another for this project.

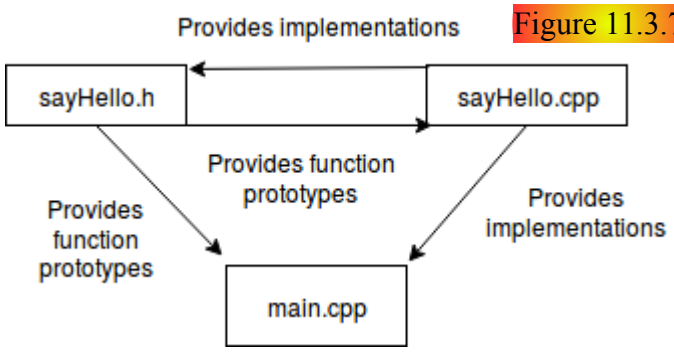
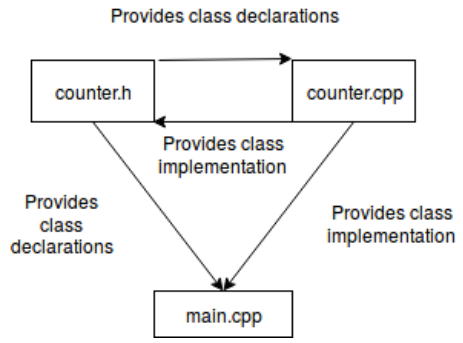



Figure 11.3.7

The counter object from the chapter on objects is another good example of how code can be split into multiple files. The original code is shown in figure 11.3.9, and will be divided similarly to the last example. The code that uses the counter will be placed into *main.cpp*, while the declarations will go into a header file called *counter.h*. The method implementations for the header will then go into a source file called *counter.cpp*.

The diagram in figure 11.3.8 shows how these files are related and what each file provides to the others.

Figure 11.3.8




```
1   #include <iostream>
2   using namespace std;
3   //Create a class called Counter.
4   //This acts as a type within the program
5   class Counter{
6     //This is the private access specifier.
7     //Everything in this section can only be accessed
8     //by code within the object.
9     private:
10      int counter;
11    //This is the public access specifier.  Anything
12    //within this section can be accessed outside
13    //of the class.
14    public:
15      //The constructor function for this class.
16      Counter(){
17        counter = 0;
18      }
19
20    //Increments the counter variable
21    void increment(){
22      counter++;
23    }
24
25    //Returns the value of the counter
26    int getValue(){
27      return counter;
28    }
29  };
30
31  int main(){
32    //Instantiate an object of type Counter
33    //This is exactly like declaring a variable
34    Counter myCounter;
35
36    //Call the increment method on the object
37    //by using the dot operator followed by
38    //the method name.
39    myCounter.increment();
40
41    //Call the getValue method by using the dot
42    //operator followed by the method name
43    cout << myCounter.getValue() << endl;
44
45    return 0;
46  }
```

Figure 11.3.9

The code is divided into *main.cpp*, *counter.h*, and *counter.cpp* as shown below.

**main.cpp**

```
1  #include <iostream>
2  using namespace std;
3
4  #include "counter.h"
5
6  int main(){
7    //Instantiate an object of type Counter
8    //This is exactly like declaring a
9    //variable
10   Counter myCounter;
11
12   //Call the increment method on the
13   //object
14   //by using the dot operator followed by
15   //the method name.
16   myCounter.increment();
17
18   //Call the getValue method by using the
19   //dot operator followed by the
20   //method name
21   cout << myCounter.getValue() << endl;
22
23   return 0;
24 }
```

Figure 11.3.10

**counter.h**

```
1  //Create a class called Counter.
2  //This acts as a type within the program
3  class Counter{
4    //This is the private access specifier.
5    //Everything in this section can only
6    //be accessed  by code within the
7    //object.
8    private:
9      int counter;
10   //This is the public access specifier.
11   //Anything within this section can
12   //be accessed outside
13   //of the class.
14   public:
15     //The constructor function for this
16     //class.
17     Counter();
18
19     //Increments the counter variable
20     void increment();
21
22     //Returns the value of the counter
23     int getValue();
24 };
```

Figure 11.3.11

**counter.cpp**

```
1  #include "counter.h"
2
3  Counter::Counter(){
4    counter = 0;
5  }
6
7  //Increments the counter variable
8  void Counter::increment(){
9    counter++;
10 }
11
12 //Returns the value of the counter
13 int Counter::getValue(){
14   return counter;
15 }
```

Figure 11.3.12

## Section 11.4: Header Guards

The *counter.h* and *counter.cpp* files from the last example form a unit. By including the *counter.h* file, counter objects may be used in other parts of the project. However, a problem occurs if the counter object is included in more than a single file. Add the following file named *anotherFunction.cpp* to the project and add the code in figure 11.4.1.

**anotherFunction.cpp**

```
1   #include “counter.h”
2
3   void myFunction(){
4     //Create a counter
5     Counter myCounter;
6
7     //Increment counter
8     myCounter.increment();
9
10    //Display value in counter
11    cout << myCounter.counterValue() << endl;
12  }
```

Figure 11.4.1

Since the counter object is used, the class declaration from *counter.h* is included. In addition to adding this file, go into *main.cpp* and add lines 5 and 6 as shown below.

**main.cpp**

```
1   #include <iostream>
2   using namespace std;
3
4   #include "counter.h"
5   //This line and the line below were added.  Nothing else changed
6   #include “anotherFunction.cpp”
7
8   int main(){
```

Figure 11.4.2

Since this file was included, the **myFunction** function can now be called within **main**. Including a source code file using #include is bad practice and should not be done. This can be seen when the project is compiled and results in an error.

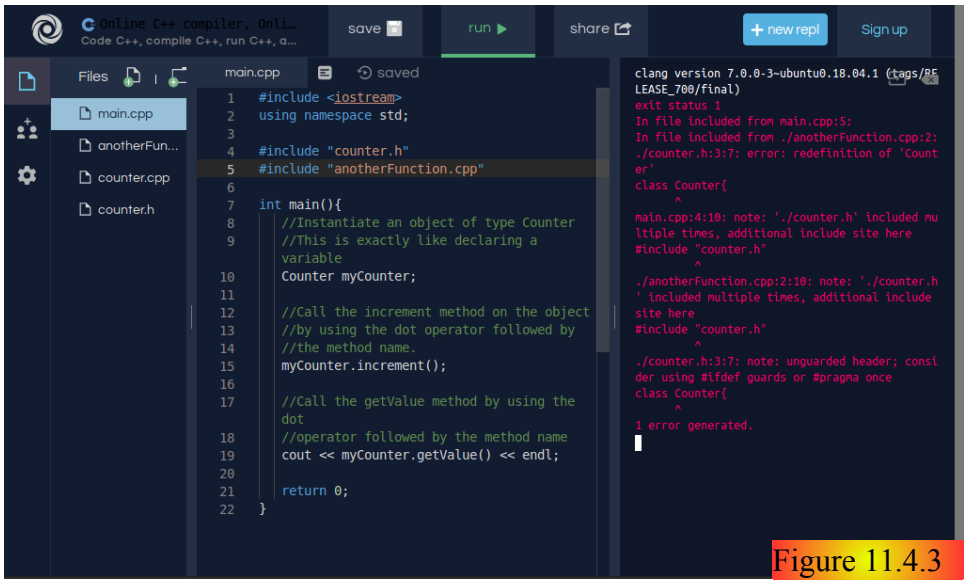


Figure 11.4.3

Based on the error message, the Counter object is declared multiple times within the project, which is not allowed. Even through the declaration is written once in the *counter.h* file, including the header into multiple files causes the declaration to be copied multiple times.

To understand how this is happening, consider the code in figure 11.4.4 and 11.4.5. Open a new tab for repl.it and create the new project with these two files.

**main.cpp**

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    #include “helloworld.cpp”
6
7    return 0;
8  }
```

Figure 11.4.4

**helloworld.cpp**

```
1  cout << “Hello, World!” << endl;
```

Figure 11.4.5

When compiled, the #include directive on line 5 in *main.cpp* will evaluate to the content of *helloworld.cpp*. Effectively, including a file with the #include directive copies the content of the included file. Because of this, the two directives at the start of *main.cpp* in figure 11.4.6 evaluate to the code in figure 11.4.7.

```
1  #include <iostream>
2  using namespace std;
3
4  #include "counter.h"
5  //Added this line
6  #include “anotherFunction.cpp”
7
8  int main(){
9  ...
```

Figure 11.4.6

The expansion of line 6 In figure 11.4.6 shows how the Counter declaration is copied multiple times. By expanding this line, the content of *anotherFunction.cpp* is copied over. One of the lines happens to be “#include “counter.h”” which means that this line will appear twice in *main.cpp*. Each of these lines expands to the content of the header file, which contains the class declaration. Because of this, the declaration of the counter object is included twice within *main.cpp*, and causes the error seen before.

This problem may be solved by using header guards, which are preprocessor directives that can be used to ensure only one copy of the class declaration is included. To add a header guard, modify the *counter.h* file as shown in figure 11.4.8.

All lines that begin with a “#” are preprocessor directives. On line 3, the directive checks if the name COUNTER_H has been defined.

If it has not been defined, the symbol COUNTER_H is defined and the class declaration is included. If the file is included a second time, the condition on line 3 evaluates to false and all of the code within the header file is not included. Because of this, the declaration can be included at most once.

```
1  #include <iostream>
2  using namespace std;
3
4  #include "counter.h"
5  //The line #include “anotherFunction.cpp”
6  //was replaced with the following
7  #include “counter.h”
8
9  void myFunction(){
10   //Create a counter
11   Counter myCounter;
12
13   //Increment counter
14   myCounter.increment();
15
16   //Display value in counter
17   cout << myCounter.counterValue() << endl;
18 }
19
20 int main(){
21 ...
```

Figure 11.4.7

```
1  //If the symbol COUNTER_H has not been
2  //defined ...
3  #ifndef COUNTER_H
4
5  //Then define it and include the counter
6  //declaration below.  If the symbol has
7  //already been declared, then everything
8  //below is not included.
9  #define COUNTER_H
10
11 //Declaration for a counter object
12 class counter{
13   private:
14     int counterVar;
15   public:
16     //These will be implemented in
17     //the counter.cpp source file
18     counter();
19     void increment();
20     void decrement();
21     int counterValue();
22 };
23 #endif COUNTER_H
```

Figure 11.4.8

An alternative preprocessor directive that can be used is **#pragma once** as shown in figure 11.4.9. This form is simpler to use and performs the same job as the header guard from the last example. Since this method is shorter and less error-prone, it is the preferred method of preventing declarations from being included twice.

```
1  //Same as using the header guards,
2  //only a lot shorter
3  #pragma once
4
5  //Declaration for a counter object
6  class counter{
7  ...
```

Figure 11.4.9

## Section 11.5: Multi-file Project: Dynamic Array

In a previous chapter, a dynamic array was created as a class. Because dynamic arrays are useful, the code can be split into a header and source file to be used by multiple parts of a project. In exactly the same way as the counter object, the dynamic array will be split into a main program, a header to hold the class declaration, and a source file to hold the method implementation.

**DynamicArray.h**

```
1  //This instructs the preprocessor to only
2  //include this file a single time
3  #pragma once
4
5  //Create a class for a dynamic array
6  class DynamicArray{
7    //These variables should not be accessed
8    //by anything outside of the object.
9    private:
10     //A pointer to a set of integers
11     int *elements;
12
13     //The amount of memory allocated for the elements.
14     int elementsSize;
15
16     //The next free space to place an element
17     int numberItems;
18   public:
19     //Constructor for a new DynamicArray
20     DynamicArray(int startSize);
21
22     //Returns the number of elements in the array,
23     //which is numberItems
24     int getSize();
25
26     //Adds a new integer to the end of the dynamic array
27     void append(int newNumber);
28
29     //Returns the integer at the given index
30     int getAtIndex(int index);
31
32     //Assigns the value to the given index in elements
33     void setAtIndex(int index, int val);
34 };
```

Figure 11.5.1

**DynamicArray.cpp**

```
1  #include “DynamicArray.h”
2
3  //Appends the new number to the end of the dynamic
4  //array and resizes if needed
5  void DynamicArray::append(int newNumber){
6    //If there is enough space free...
7    if( numberItems < elementsSize){
8      //...append to the end of the array and set the next index as free
9      elements[numberItems] = newNumber;
10     numberItems++;
11   }else{
12     //We have run out of space, so the array needs
13     //to be re-allocated.
14
15     //Save a temporary pointer to the old array
16     int *temp = elements;
17
18     //Allocate twice the size for the new array
19     elements = new int[elementsSize*2];
20
21     //Copy over elements from the old array to the new array
22     for(int i = 0; i < elementsSize; i++){
23       elements[i] = temp[i];
24     }
25
26     //The size of the new array is twice the old size,
27     //so double elementsSize and free the old array
28     elementsSize *= 2;
29     delete [] temp;
30
31     //Now the new element can be placed
32     elements[numberItems] = newNumber;
33     numberItems++;
34   }
35 }
36
37 //Returns the integer at the given index, else
38 //0 if the index is out of range
39 int DynamicArray::getAtIndex(int index){
40   if( index < numberItems ){
41     return elements[index];
42   }else{
43     return 0;
44   }
45 }
```

Figure 11.5.2

**DynamicArray.cpp**

```
46  DynamicArray::DynamicArray(int startSize){
47     //First free index is at index 0 to start
48     numberItems = 0;
49
50     //Set the start size
51     elementsSize = startSize;
52
53     //Allocate an integer array to hold elements
54     elements = new int[startSize];
55  }
56
57  //Returns the number of elements held in the array
58  int DynamicArray::getSize(){
59     return numberItems;
60  }
61
62  //Sets the given element in the array if the index
63  //is within range
64  void DynamicArray::setAtIndex(int index, int val){
65     if( index < numberItems ){
66        elements[index] = val;
67     }
68  }
```

Figure 11.5.2 (cont.)

main.cpp

```
1  #include <iostream>
2  #include “DynamicArray.hpp”
3  using namespace std;
4
5  int main() {
6    //Create a dynamic array with an initial capacity
7    //of one element. This line envokes the constructor
8    //which initializes the dynamic array.
9    DynamicArray a(1);
10
11   //Append a series of elements to the array
12   a.append(10);
13   a.append(11);
14   a.append(12);
15   a.append(13);
16   a.append(14);
17
18   //Use the dynamic array's methods to print out its content
19   for(int i = 0; i < a.getSize(); i++){
20     cout << "Element #" << i << " is " << a.getAtIndex(i) << endl;
21   }
22
23   return 0;
24 }
```

Figure 11.5.3

When constructing a multi-file project, the key is understanding that code goes into source files and declarations go into header files. In addition, executable code should never be directly included within a program using the #include directive.