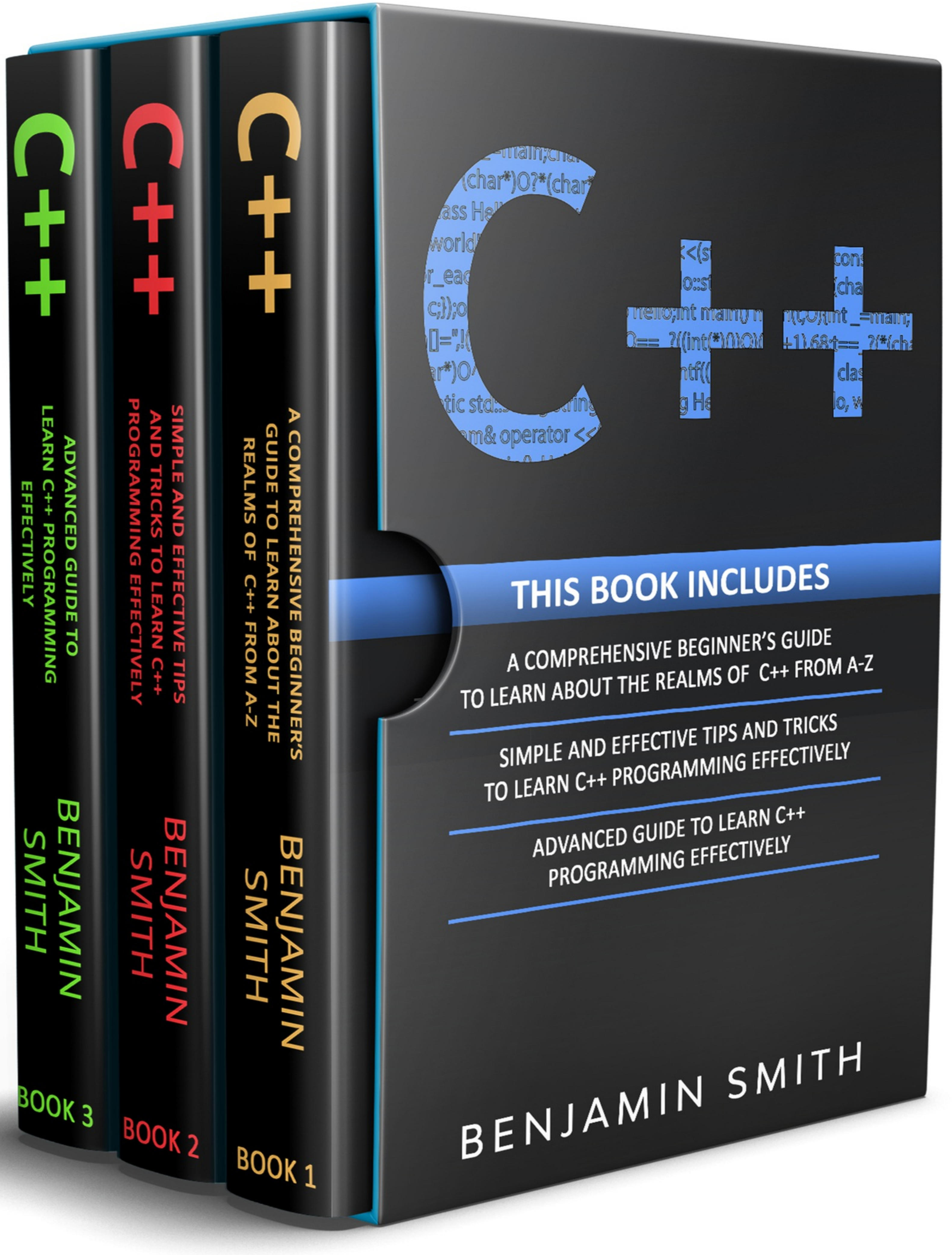

C++
THIS BOOK INCLUDES
A COMPREHENSIVE BEGINNER'S GUIDE TO LEARN ABOUT THE REALMS OF C++ FROM A-Z
SIMPLE AND EFFECTIVE TIPS AND TRICKS TO LEARN C++ PROGRAMMING EFFECTIVELY
ADVANCED GUIDE TO LEARN C++ PROGRAMMING EFFECTIVELY
BENJAMIN SMITH
C++
A COMPREHENSIVE BEGINNER'S GUIDE TO LEARN ABOUT THE REALMS OF C++ FROM A-Z
BENJAMIN SMITH
BOOK 1
C++
SIMPLE AND EFFECTIVE TIPS AND TRICKS TO LEARN C++ PROGRAMMING EFFECTIVELY
BENJAMIN SMITH
BOOK 2
C++
ADVANCED GUIDE TO LEARN C++ PROGRAMMING EFFECTIVELY
BENJAMIN SMITH
BOOK 3

C++

BENJAMIN SMITH

Table of Contents

C++
Simple and Effective Tips and Tricks

to learn C++ Programming Effectively

C++

A Comprehensive Beginner’s Guide to Learn About the Realms of C++ From A-Z

BENJAMIN SMITH

Introduction

C++ is a computer programming language, and as from one perspective, a computer program language is a sequence of instruction fed into a computer that dictate to the computer what to do. These instructions are executed through the flow of electrical impulses that affects the computer’s memory through interaction with the input and output devices. There are different types of computer programs playing different roles; one program might let a computer perform the role as a financial calculator, while another computer program could transform the computer into a worthy chess opponent.

In the mid-1980s, Bjarne Stroustrup of AT and T Bell Labs developed the C++ programming language. C++ is an extension of the previously developed C programming language by AT and T Labs in the early 1970s. Originally, AT and T developed the C programing language to write UNIX operating systems, system-level software, and embedded system development. Initially, following after the development of the C++ programming language, it provided object-oriented programming features, and later on, generic programming capacities were added.

The C++ programming language is useful in both commercial and industrial software development. C++ is a powerful programming language used for developing complex engineering, sciences, and business systems. Some common software written in C++ include Adobe Creative Suites, macOS, Microsoft Office, Microsoft Windows 8, and so on. The C++ programming language is complex itself so that it can meet up with the needs of commercial software development. Experienced software developers can accomplish great things with C++, but beginners may have a difficult time with it because it is closer to machine language than human language.

We’re keen on understanding in this book, and so, to fully understand everything you need to know about C++ programing language, there are some development tools you need to acquit yourself to how they work.

This book is not aimed to cover all facets of the programming language, but to guide you on all, you ought to know about the C++ programming language. Despite the plethora of things you need to know about C++, this book will bring as much of it into ten concise chapters letting you learn all you need to know about C++ from
A-Z.

Chapter 1

Writing a C++ Program

C++ has a particular structure of writing them. The syntax of the code much be correct. If not, the compiler would give an error message, and it would not be executable. In this chapter, we shall be introducing C++ with simple executable examples you can try. Most C++ codes are executable in a lot of other programming languages like C, Ada, C#, and Java but with a slightly different syntax.

The General Structure of a Simple C++ Code

Below is a one of the simplest C++ code that does something:

```
#include <iostream>
int main() {
        std::cout << “General structure of a simple C++ code!\n”;
}
```

The above code is what a simple C++ code looks like. You type it into an editor, and you run it through a compiler, the program will print the message:

General structure of a simple C++ code!

This simple C++ code above is a four non-blank lines of code which includes:

- #include <iostream>

This coding line is a preprocessing directive. The preprocessing directive in the C++ source code always starts with a **#** symbol. The symbols direct the preprocessor, so it adds a predefined source code to an existing one before compiling the process. This process is automatic. The **iostream** in this line means we want to use an object from the **iostream** library. The iostream is a collection of pre-compiled C++ codes executable by the C++ program. The **iostream** library also contains elements that handle the input and output (I/O) like getting information from the keyboard, dealing with files, and printing a display on the screen.

In the example, **std::cout** is not a part of the C++ code itself. It is one of those things related to output and input the C++ compiler develop and store in the **iostream** library. Now the compiler needs to be aware of these **iostream** items so it can compile the program. So, that is why the **#include** directive is used to specifies a file called a header, which contains the library code specifications. So, the compiler would use the **std::cout** to cross-check the specification in the **<iostream>** header. A lot of C++ programs make use of the **#include <iostream>** directive, although there are other programming codes headers as well.

- int main () {

This programming line specifies the actual beginning of the program. In this line, we're declaring a function named main. Any C++ program containing this function is executable. The int means integer, which is a fundamental variable a compiler used to assign numeric variables with whole numbers. The opening of the curly brace at the end of the line is the beginning of the main program. It's a rule of thumb that the body of a function must contain the statement of the function to execute.

- std::cout << "General structure of a simple C++ code!\n";

The body of the main function in this code is only one statement. The statement is an executable program that directs to print the message "*General structure of a simple C++ code!"* on the screen. A statement simply means a fundamental unit of execution in the C++ program. Functions, for example, contains statements a compiler can translate into executable machine language. There are so many kinds of statements which we will explore more in the next chapter. Take note that statements in C++ ends with a semicolon (;).

- }

The closing curly brace is what programmers use to mark the end of a particular function. So, both the opening and closing curly brace are needed for defining every function.

Editing, Compiling and Running a Program

When it comes to the best development environment, C++ programmers have two options. The first option includes the use of a command-line environment, together with a collection of independent tools. The second option includes the use of an IDE like Virtual Studio, which groups the compilation steps into a process called build. These powerful IDEs have a myriad of features and configuration options that can be bewildering to those still learning how to program. In a typical command-line environment, the developer only needs to type in some simple commands in the console window to edit, compile, and eventually execute the program. Some programmers prefer the flexibility and simplicity of a command line of build environments, most especially in less complex projects.

An example of a common command-line building system is the GNU Compiler Collection (GCC). The GCC C++ compiler is one of the most conforming compilers available for C++ standards. The GCC++ is supported by Linux, Apple Mac, and Microsoft Windows platforms, and it is a free, open-source software. The following are important tools that useful in editing, compiling, and running a program:

- **Editors**

An editor is a tool that allows software developers to enter the program source code and save it into files. Most editor tools out there enhances the programmer's productivity, such that it helps highlight language features with colors. Programmers must follow the strict syntax rules when creating computer programs. The syntax of a language is the way the different pieces and components of the sentence are arranged to form a sentence the computer would understand.

For instance, the sentence "the short girl runs quickly to the window" makes use of proper English syntax. And by comparison, the sentence "girl the short runs window to quickly the" does not make use of proper English syntax and so it's not correct. In like manner, if programmers don't follow the syntax rule of programming, there are bounds to be errors, and the code wouldn't be acceptable to be compiled and executed. Some syntax-aware editors make use of special annotations and colors to alert the developer of a syntax error before it is compiled.

- **Compilers**

A computer compiler is a tool used to translate a source code to a target code, which may be a machine language for an embedded device or a particular platform. It could also be that the target code is another machine language. In the earliest versions, C++ compiled source codes into C. Then, a C compiler would process the C code to produce an executable program. But today, C++ compilers translate the source code into machine language. The C++ complete set of building tools includes a linker, preprocessor, and compiler.

- **Preprocessor**

A preprocessor is a building tool used to modify or add contents to the source file before starting the compiler to process the code. Mainly, developers use the preprocessor to #include information.

- **Compiler**

The compiler compiles the source code to machine language, as described above.

- **Linker**

The linker acts as a link that combines the machine code the compiler generates with the compiled code or library code from another source to create a complete executable program. Most times, a C++ compiled code can’t run on its own; it often requires some additional kind of machine code for it to be an executable program. That missing machine code is often precompiled and stored in a library.

Generally, don’t think of the preprocessor, compiler, and liner as separate programs, but as one process taking place to translate source codes to executable programs.

- **Debuggers**

The debugger is another building tool programmers use to trace a program’s execution to locate and correct any errors made in the program’s implementation. With the help of the debuggers, programmers can run a program and see the line in the source code, causing the current action simultaneously. A programmer can watch the variable's values and other programming elements in case their values change as they ought to. Debuggers are useful in locating errors or bugs and for fixing programs containing the errors.

- **Profilers**

The profiler helps a programmer collect statistics of a program's execution, which can help the programmer fine-tune appropriate parts of the code to improve the overall performance. The profiler helps to show how many times and how long a part portion of a program was executed in a particular run. Furthermore, profilers are useful in testing purposes so that all the codes have a use somewhere during the test, which is also known as coverage. The main use of the profiler is to locate those parts of a program that needs improvement so that the program can run faster.

Variations of Writing Our Simple Program

The example below shows an alternative way to write the simple example we created above.

```
#include <iostream>
using std::count;
int main () {
        cout << "General structure of a simple C++ program!\n";
}
```

The example above made use of a using directive, which allows us to make use of a shorter name for the **std::cout** printing object. In fact, we could even omit the prefix **std::** and use the shorter name **cout**. However, this directive is optional, and if we're to omit it, then we must make use of the longer name. The name **std** means "standard," and in the code, the prefix **std** tells us that **cout** is a part of a collection of names known as the standard namespace. The standard namespace holds names for all functions and types of standard C++ that are needed for all standard-conforming C++ developing environment. Outside the standard library, components provided by thirds party programmers reside in their own separate namespaces. Below is another way to write a code using the shorter name for **cout** within the C++ program.

```
#include <iostream>
using namespace std;
int main () {
      cout << “General structure of a simple C++ program!\n”;
}
```

In the example above, you’d notice the use of the blanket **using,** which makes all the names in the **std** namespace available for the compiler to use. Using this approach above has some kind advantages for smaller programs, as the blanket **using** directive allows developers to make use of shorter names. The blanket **using** directive also makes use of fewer lines of code, especially when the program makes use of multiple elements from the **std** namespace.

Your choosing to use the **using** directive doesn’t have any effect on the final product – the executable program. There are three versions of the using directive, blanket using, focused using, and no using, and compilers generate the same machine language code for all three. So, we can choose any one we prefer as long as it enhances our ability to manage and write the software project properly.

However, it is important to note that even though the blanket **using** approach has its place amongst the three other **using** directive, it is often discouraged when writing a complex software project. Later in this book, we would explain in full the disadvantages of the blanket **using namespace std** directive when you have enough experience with the C++. So, for now, we’d try as much as we can to avoid the blanket **using** statements as we strive for the best practices.

Chapter 2

Variables and Values

In this chapter, we shall be digging deep into some building blocks of C++ program. We would be experimenting with concepts like declarations, reserved words, assignments, numerical values, identifiers, and variables.

Integer Values

Integers are numeric values that are whole numbers. So numeric values in factional parts is not an integer value in C++ code. For example, five (5) is an integer. Furthermore, the integer can either be zero, negative, or positive. Other examples of integers include 6, - 13, and – 2006, but 5.4 is not an integer because it is not a whole number. Take a look at Code 2.1 below as it shows how an integer value can be employed in C++ code:

Code 2.1

```
#include <iostream>
int main () {
        st::cout << 5<< ‘\n;
}
```

If you notice, unlike the other codes we wrote earlier, this code does not make use of quotation marks (“). The number 5 will still appear even when there are no quotes. Also, the expression “**\n**” signifies a single newline character. But if you’re writing code with multiple characters comprising of a string, then you need to use double quotes (“). In C++, a single character is enclosed in single quotes (‘) and represents a distinct type of data. Check out this example below:

Code 2.2

```
#include <iostream>
```

```
int main () {
        std::cout << “5\n”;
}
```

Code 2.1 and Code 2.2 may seem similar, but they are quite different. Code 2.1 prints the value of number five, while Code 2.2 above prints a message containing the digit five. The difference here may seem unimportant, but in later in this chapter, we’d show you how the presence and absence of quotes in a code can make a big difference in its output. In the statement std::cout << “5\n”; sends a message to the output stream, which is the string **“5\n”**. On the other hand, the statement std::cout << 5 << ‘\n’; sends two messages to the output string; the first is the integer value 5, and the second is the newline character **‘\n’**.

In some cases, developers could write the same Code 2.2 in a different way:

Code 2.3

```
std::cout << 5 << std::endl;
```

Now, even though Code 2.2 and Code 2.3 behave exactly alike, they do not mean exactly the same thing. The **std::endl** expression involves a newline character, and it also performs additional work that isn’t necessary. Programs meant for significant printing would execute faster if you end their output line with **‘\n’** rather than **std::endl**. However, we can ignore the difference in speed when we’re printing on the console, but the difference in speed is not negligible when you’re printing on other output streams or files. Nonetheless, we’ve been making use of the **‘\n’** most times for printing newlines because it is a good habit to form as a beginner as it involves only a few keystrokes. Likewise, the three major modern computing platforms; Apple macOS, Microsoft Windows, and Linux, handle newlines differently.

Also, in C++ code, integers cannot contain commas. In other words, we can write the numbers nine thousand, five hundred and twenty-four as 9524 and not 9,524. Modern C++, however, supports single quotes (‘) to separate digits. So, you can write the same number as 9’524 without having an error, and it would also improve human comprehension when reading larger numbers in the C++ code.

Mathematically, integers are not unbounded, which means integers are infinite. But in C++, integers have limits because computers have a finite amount of memory. So, the maximum range of integer supported depends on the C++ compiler and the computer system. If, for example, we exceed the range of integer the C++ compiler can read, what happens? Try Code 2.4 on your 32-bit computer system and see what happens.

Code 2.4

```
#include <iostream>
int main () {
       std::cout << - 9000000000 << ‘\n’;
}
```

According to the range of integer, a 32-bit can support, nine billion is too large for it, so it would either display an error or a wrong output. Whenever you get a warning in a code, it indicates that there is a potential problem, but it doesn’t stop the compiler from proceeding to give an executable program. So, as a programmer, you need to heed to warnings because the execution often produces a meaningless output.

Other programming languages as well have a limited range of vale because each number is stored in a fixed amount of memory. Meaning, if you want to save a larger number, then it would take up more storage space in the memory. To infinite a set of mathematical integers for C++, then you need an infinite amount of memory.

Variables and Assignment

Just like in algebra, variables represent numbers, and this same principle is applicable in C++ only that C++ variables can accept values other than numbers. Code 2.5 shows how C++ used variables to store an integer and also prints the value of that variable.

Code 2.5

```
#include <iostream>
int main () {
      int a;
      a=5;
      std::cout << a << ‘\n’;
}
```

There are three statements in the main function in Code 2.5:

- int a;

When writing a C++ code, all variables must be declared. And the **int a;** is an example of a declaration statement. A declaration statement specifies what type of variables is being used. In Code 2.5, the **int** indicates that the variable being used there is an integer and that the name of the integer is **a**. So, we can read the code as the variable **a** has type **int**. C++ code supports other types of integers, some type of integer require more or less memory space for storing the variable's value. A declaration statement allows the compiler to know if the developer is using the variable in the right way in the program. For example, the declaration statement can let us see if the integers are addable, just like in mathematics. In some other data types, addition is not possible, so it's not allowed. The compiler also ensures the variables involved in a typical addition process are compatible with the addition rules. It would report an error if it is not compatible.

The compiler will also report an error is you attempt to use a variable without a declaration statement. The compiler can't verify the variable's proper usage, and it cannot deduce the storage requirement if it is not declared. When you use a declaration statement to declare a particular variable, you can't redeclare that variable within the same context.

- a=5;

This line is an assignment statement. You use an assignment statement to associate a value to a variable. The symbol = in an assignment statement is key, and it's also known as an assignment operator. In Code 2.5, the value 5 is being assigned to the variable a, meaning the value 5 will be stored in the memory the compiler reserved for the variable named **a**.

- std::cout << a << '\n';

This statement simply means the current value of **a** would be printed. Take note of the lack of quotation marks here; it's very important. Consider Code 2.6 and Code 2.7:

Code 2.6

```
    std::cout << a << '\n';
```

Code 2.7

```
std::cout << “a” << ‘\n’;
```

In the statement in Code 2.6, it prints 5 as the values of the variable a, but in the statement in Code 2.7, it prints **a** as the message containing the single letter **a**.

In Code 2.5, the assignment operator (=) doesn’t mean the same thing as the equality operator in mathematics. The = operation in Math means that the expression on the left is equal to the expression on the right. But in C++, the = makes the variable on the left take the value of the expression on the right. It’s best to read a=5 as “**a** is assigned the value of 5.” It's important we point this out, because the equality operation is symmetric in mathematics, meaning a=5 and 5=a, but in C++ it is not so.

Code 2.8

```
5=a
```

Code 2.8 means you’re trying to reassign a value to the literal integer value 5, and it’s impossible because 5 is always 5, and you can’t change it. Such a statement in a compiler would issue an error.

Code 2.9

```
error C2106: ‘=’ : left operand must be 1-value
```

Variables can also be reassigned to different values if there’s a need for it.

Code 2.10

```
#include <iostream>
int main () {
    int a;
    a=1;
    std::cout << a << ‘\n’;
    a=2;
    std::cout << a << ‘\n’;
    a=3;
    std::cout <<a<< ‘\n’;
}
```

If you take a close look at Code 2.10, each print statement is identical, but after running the program, you’d get different results. Variables can also be given values the time you declare it. Take, for example, Code 2.11:

Code 2.11

```
#include <iostream>
int main () {
        int a=5;
        std::cout << a << ‘\n’;
}
```

If you notice, in this code, the declaration and assignment of the variable, **a** is done in the same line statement and not two, as we have explained previously. The combination of declaration and assignment is called initialization. Likewise, C++ also support another syntax for initializing variables as should in Code 2.12:

Code 2.12

```
#include <iostream>
int main () {
        int a {5};
        std::cout << a << ‘\n’;
}
```

Code 2.12 is another way of combining declaration and assignment, but it is not common, especially for simple variables. Code 2.12 is necessary when you want to initialize a more complicated kind of variable called objects. We’d talk more about objects in later chapters of this book.

It is also possible to declare multiple variables of the same type in a single statement if desired. Consider the statement in Code 2.13:

Code 2.13

```
int a, b, c;
```

The statement in Code 2.13 declares three integer variables. Also consider the next statement that follows Code 2.13:

Code 2.14

```
int a = 1, b, c = 3;
```

In Code 2.14, b is undefined, and so the declaration can be split into multiple declaration statements:

Code 2.15

```
int a = 1;
int b;
int c = 5;
```

In Code 2.15, a multiple declaration statement, the type name int must appear in each statement.

Furthermore, a compiler maps a variable to a location in your computer's memory. Take a look at Code 2.16:

Code 2.16

```
int x, y;
x = 1;
y = 2;
x = y;
y = 3;
```

Importantly, if we consider the statement **x = y**, it doesn't mean **x** and **y** are saved in the same memory location. It simply means that the same value that is in **x** has been copied to **y,** but they still have different memory locations. Going back to Code 2.16, **x** was first declared as 1. Also, **y** was later declared as 2 in its own memory location. When **x** was declared the same value as **y** (**x = y**), the original value in **x** was overwritten with the value in **y** in that statement.

Identifiers

Unlike in mathematics, where variables are given one letter names like a, in programming, developers should use longer and more describes variable names as in **user_name, sum,** and **altitude**. When giving a variable a name, it must be related to the purpose of the program. A good variable name makes the program more readable by humans. Generally, programs contain so many variables, so a well-chosen variable name renders the program more understandable.

C++ programming has a very strict rule for naming variables. Naming variables is a perfect example of an identifier. Identifiers are words used to name things like variables. Identifiers can also name other things like classes and functions. Identifiers have the following forms:

- It must contain at least one character.
- Its first character must be an alphabetic letter, either upper or lower case or the underscore.
- Its remaining characters can be either an alphabetic character (upper or lower case), an underscore, or a digit.
- No other characters are permissible, including spaces.
- Reserved words can’t be used as an identifier.

Here are some typical examples of acceptable and unacceptable identifiers.

- The following are acceptable identifiers and can be used to name variables: a, a2, USB, E6400, and flow_22
- The following are unacceptable identifiers: mini-port, last entry, 4all, #1, and class (class is a reserved word).

Some types of programming languages do not require you to declare a variable; it declares it automatically. Such programming languages assume the different types of variables based on how you use it in the different sections of the program. These types of languages are known as dynamically-typed languages. But the C++ programming language is statically-typed. And a statically-typed language is a type of programming language where the variable must be explicitly specified before a statement in a program can use it. The idea of having to declare all variables in C++ might seem trying at first, but it offers several advantages:

- In a statically-typed language where variables must be declared, the compiler can easily catch the typographical errors that dynamically-typed language can’t detect.
- Also, in statically-typed language where variables must be declared, the compiler can catch invalid operations that dynamically-types

variables can’t detect. For example, you may want to declare a variable to be of type **int**, but you accidentally assign it a non-numerical value to the variable.

- Ideally, because the C++ program requires you to declare variables, it makes you plan ahead and think more about the variables your program may need. The purpose of every variable is tied to its type, so you need to have a clear notion of the purpose of variables before declaring it.
- Generally, statically-typed languages are more efficient than dynamically-typed languages.

In addition, C++ is case sensitive and so capitalization matters a great deal. For example, the word **if** is reserved, but the words iF, IF, or If are not reserved and so you can use them to name variables. In the same manner, identifiers are case sensitive, and so, a variable called **name** is different from a variable called **Name**. Giving variables names that are distinguishable by capitalization is confusing.

Additional Integer Types

There are several other different types of integers supported by the C++ language. The type **short int**, meaning short, represents integers that doesn’t occupy much bytes of memory like the int type. In retrospect, the smaller the memory space, the integer type occupies, the smaller the range of integer value. Standard C++ coding requires the **short** type to be smaller than the **int** type, as a matter of fact, they could represent the same set of integer values. There is also the **long int** type, which can also be written as just **long**. In this case, the **long** type can occupy more space than the **int** type, allowing it to represent a much larger range of values. Standard C++ coding requires the **long** type to be bigger than or equal to the **int** type. Lastly, there is the **long long int** type also written as **long long** for short. The **long long** type may be larger than a **long**. The relative ranges of values hold in C++ standards is as follows:

short int ≤ int ≤ long int ≤ long long int

Floating-Point Types

A lot of the computational task we do requires numbers, mostly numbers with fractional parts. For instance, a formula for mathematics that computes the area of a circle, given the circle's radius, will involve the use of the value of π, which is approximately 3.14159. The C++ language supports such non-integer numbers, and they are known as floating-point numbers. The name comes from the ideology that in mathematical calculations, decimal points can float to various positions within the number to obtain significant digits. There are different types of floating-point numbers; the type **double** and **float** are the two most common.

The type double is often used, and they represent double-precision floating-point. It can also represent a wider range of values with more digits of precision. On the other hand, the float type represents single-precision floating-point values, which are less precise.

Constants

What are constants? Take, for example, the speed of light or Avogadro's number; those are scientific constants. Constants are degree of precisions that have been calculated and measured, and they don't vary. In C++, constants have the same meaning. They are declared just like the way we declare variables by adding the keyword **const**. Once you declare a constant and initialize it, it can be used like a variable, expecting to reassign a constant. It is an error to have a constant on the left side of the assignment operator outside the declaration statement. For example:

```
PI = 9.8;
```

This would cause the compiler to run into an error and may display something like

error C3892: 'PI'; you cannot assign a variable that is const

and it would fail obviously to compile the program. Because scientific constant doesn't change, consider code 2.17 below:

Code 2.17

```
#include <iostream>
int main () {
const double c = 2.998e8, avogradros_number = 6.022e23;
std::cout <<"Speed of light = " << c << '\n;
std::cout << "Avogadro's number = " << avogrados_number << '\n';
}
```

Because it is not possible to assign a constant outside of a declaration statement, all constant has to be initialized where they are declared. So, generally, C++ developers have to express constant names in all capital letters. By doing so, it makes it possible for the human readers to distinguish between a variable and a constant easily.

Other Numeric Types

C++ also supports other numeric types which include:

1. long int: This typically provides integers that have a greater range than the **int** type, and it has the long abbreviation. This form of numeric date guarantees to provide a range of integer's values that are large at least up to the **int** type. Take, for example, an integer with L suffix in **19L** has type **long**. You can also use a lower case l as a suffix, but done use it too frequently because it's hard for human readers to distinguish between lower case l and digit one 1.

2. short int: This typically provides you with an integer value that has a smaller range than the int type, and it's abbreviated **short**. This numeric form guarantees that the range of **int** is bigger than the range of **shorts**.

3. unsigned int: This numeric data type is typically restricted to nonnegative integers, and its abbreviated name is **unsigned**. The **unsigned** type is limited in nonnegative values, and it represents twice as many positive values as in the **int** type.

4. long double: They can extend their precision and range of the **double** type.

The C++ language specifies the minimum precision and ranges for all numeric data types, particularly the C++ compiler, which may exceed the specified minimum. So, C++ provides these varieties of numeric types for specialized purposes that are related to building highly efficient programs. Although we'd have very little use of these types because most of our examples will be making use of mainly the numeric types **int** for integer, **double** for an approximation of real number and **unsigned** when nonnegative integer values are needed.

Characters

Characters abbreviated as **char** is an example of a data type used to represent single characters – letters of the alphabet, either lower or upper case, punctuation, digit, and control characters. Most systems support the use of the American Standard Code for Information Interchange (ASCII) character sets. The ASCII standard can be represented in up to 128 different characters. In a C++ source code, the characters can be enclosed in single quotes, consider code 2.18 for more explanation.

Code 2.18

```
    char ch = ‘B’;
```

The standard double quotes (“) are reserved for strings. Strings are composed of characters but don't confuse strings and char; they are different. The following code would produce an error in a compiler.

Code 2.19

```
    ch = “B”;
```

This error is due to the double quotes used to close the character, which is not a string. So, a string can't be assigned to a character variable. C++ permits **chars** to be saved as integer values. So, you can assign a numeric value to char variables and assign characters to numeric variables. In the statement

```
    ch = 10;
```

the value 10 is the ASCII that is being assigned to the **char** variable. So is we're to print ch as in

```
    ch = 10;
    std::cout << ch;
```

The corresponding character B would be printed on the screen because ch's declare type is **char** and not **int** or some other numeric type. Consider the example in code 2.20;

Code 2.20

```
#include <iostream>
int main () {
char ch1, ch2;
ch1 = 10;
ch2 = ‘B’;
std::cout <<ch1 << “,” << ch2 << “,” << ‘B; << ‘\n’;
}
```

Also, characters and integers can be assigned freely to each other, but the range of **char** is smaller than the range of an **int**. So, be careful when assigning an **int** value to a **char** value. Furthermore, some characters are non-printable. Below are some of the examples:

- ‘’\f’: This is the formfeed character
- ‘\θ’: This is the null character used in C strings
- ‘\t’: This is the tab character
- ‘\a’: This is the “alert” character used to cause a beep sound or tone in some systems
- ‘\b’: This is the backspace character
- ‘\r’: This is the carriage return character
- ‘\n’: This is the newline character

Enumerated Types

C++ also allows developers the ability to create a new and very simple type of list in any possible value of that type. These types of values are called enumeration type or enumerated type. The enumeration type is abbreviated as **enum**. Check out the line below, it shows a simplified way to define an enumeration type:

Code 2.21

```
enum color {Violet, Blue, Green, Yellow, Orange, Red };
```

In the example above, the new type is name **Color**, while the variable type of **Color** can assume any type of color which appear on the list within the curly braces. It also follows with a semicolon after the closing curly brace. Sometimes the enumerated type can be in the format below:

Code 2.22

```
enum Color {
Violet,
Blue,
Green,
Yellow,
Orange,
Red
};
```

But in a complier, both code 2.21 and code 2.22 makes no difference.

Chapter 3

Arithmetic and Expressions

In the previous chapter, we introduced the C++ numeric type to perform arithmetic and build expression. Some other important concepts we'd be covering in this chapter include source formatting, user input, dealing with errors, and comments.

Expressions

A literal value, for example, 10 and a properly declared variable like a are examples of an expression. You can use operators to combine variables and values to form a more complex expression. Consider the example in Code 3.1

Code 3.1

```
#include <iostream>
int main () {
int value1, value 2, sum;
std::cout << “Please enter two integer values: ”;
std::cin >> value 1 >> value2;
sum=value1 + value2;
std::cout << value1 << “ + “ << value2 << “ = “ << sum << ‘\n’;
}
```

In the example above, the following stands for:

- int value1, value 2, sum;

This statement is a declaration statement, and it declares the three integer variables. But this statement does not actually initialize them. As we continue to examine the remaining parts of the program, you'd see why it would be superfluous to assign values to the variables here.

- std::cout << “Please enter two integer values: ”;

In this statement, the user is prompted to enter some information. This statement is where we often print, but we didn’t terminate it with the usual end of line marker ‘\n.’ So, when you type in the values, they appear on the same line as the message asking for the values. When you press the enter key to complete, the cursor will automatically proceed to the next line.

- std::cin >> value 1 >> value2;

This statement causes the program to stop when you enter two numbers and then press the Enter key. The first number you enter will be assigned to value1, while the second will be assigned to value2. When you press the enter key, the values entered will be assigned to the variable. The **std::cin** is an object in C++ used to read input from the user.

- sum=value1 + value2;

This is an assignment statement, and it contains the assignment operator (=). The variable sum appears on the left-hand side of the assignment operator. In other words, **sum** will get a value when the statement executes.

Every expression has a value, and a process of determining the value of an expression is called evaluation. Evaluating simple expression is easy, but things get more complex when you’re trying to evaluate a smaller expression that makes up a more complex expression. Basically, the following are the simple C++ arithmetic operators.

- % means modulus
- / means division
- * means multiplication
- - means subtraction
- + means addition

Mixed Type Expressions

Expressions can also contain mixed elements for example, consider code 3.2:

Code 3.2

```
int x = 2

doule y = 8.1, sum;

sum = x + y;
```

In the source code above, **int** proceeds to a **double,** so how's this arithmetic performed. But the range of **ints** falls within the range of **doubles,** thus letting you represent any **int** value with a **double**. Also, the **int** value 2 can be represented as a **double** 2.0. So, it is completely reasonable to assign int values to double variables. This process is called widening, and it is quite safe to widen **int** to a **double.** Consider the code 3.3 source code fragment.

Code 3.3

```
double d1;
int i1 = 100;
d1 = i1;
std::cout << "d1 = " << d1 << '\n';
```

Assigning a **double** to an **int** variable is not always possible since the **double** value cannot be in range of **ints**. The **double** variable has to also fall within the range of **ints** and not a whole number because the **int** variable can't manage the fractional part. So, if you were to write code 3.4, you'd get an error message.

Code 3.4

```
double d = 2.9;
int I = d;
```

In Code 3.4, the second line will assign 2 to i but lose the 0.9 fractional part because proper rounding can't is done. In fact, the visual C++ compiler will warn you of a potential problem with:

> *warning C4244 '=': conversion from 'double' to 'int,' possible loss of data*

In like manners, converting from a wider type to a narrower type (like **double** to **int**) is known as narrowing. In narrowing, it is necessary to assign a floating-point value to an integer variable. And if the value we want to assign is within the **range of the int**, and has no fractional parts, and pose no truncation harm, then the assignment is safe. If we want to perform an assignment and not get a warning from the compiler, then we need to use a procedure known as cast, also called typecast. The cast causes the compiler to accept the assignment without issuing a warning.

Operator Precedence and Associativity

The normal rules of arithmetic apply when there are different operators used in the same expression. All C++ operators has an associativity and precedence.

- Associativity: This is when an expression contains two operators that have the same precedence.
- Precedence: This is when an expression contains two kinds of operators.

To further understand how precedence works, consider the expression 2 + 3 * 4. The expression can be interpreted as (2 + 3) * 4, which would be 20. This same expression can be expressed as 2 + (3 * 4), which would be 14. So, which one is the correct interpretation. Just like in normal mathematical arithmetic, multiplication and division have equal importance and are usually performed before addition or subtraction. So, in the expression, multiplication is performed before addition since multiplication precedence over addition. The correct answer is 14, according to mathematical arithmetic.

The multiplicative operators (*, / or %) all have equal precedence with each other, and the additive operator (binary, +, and -) all have equal precedence with each other. The multiplicative operator has precedence over the additive operators. And just like in standard arithmetic, in C++, parentheses can override the precedence rules. Consider the expression (2 + 3) * 4, which is equal to 20.

Another way to easily go about this is to follow B.O.D.M.A.S order of operation, which interprets brackets, order, division, multiplication, addition, and subtraction, respectively. To see how associative works consider the expression 2 – 3 – 4. The two operators in this example are the same, so they have equal precedence. But if we should apply the first subtraction, before the second as in (2 – 3) – 4, our answer would be – 5. But if we should apply the second subtraction before the second as in 2 – (3 – 4), our answer would be 3. All binary operators except assignment are left-associative with the assignment being right-associative.

Consider the table below; it shows operator precedence and associativity. Operators that are in the same row have the same precedence. The operator in each of the rows has a higher precedence than the operators below it.

Arity	Operators	Associativity
Unary	+, -	
Binary	*, /, %	Left
Binary	+, -	Left
Binary	=	Right

Comments

A good programmer engages in annotating their code by using remarks to explain the purpose of a section in their code. This is why programmers choose to write a section of code in a particular way they do. These remarks are what help human readers easily understand the codes and not compilers. Choosing the right comments and identifiers to use aids in the assessment process of a program. The compiler ignores any text in the comment box. So, C++ supports two main types of comments, the single-line comment, and the block comments.

- **Single Line Comment**

This is the first type of comment that helps in writing a single line remark. Consider

```
// Compute the average of the values
Avg = sum / number;
```

The first line here explains the comment and what the statement following it is meant to do. If you also noticed, the comment begins with a double forward slash symbol (//) and continues until the end of the line. Because of the double slash, the compiler ignores the content on the rest of the line. Using this type of comment is also useful in appending a short comment at the end of a statement. For example:

```
avg = sum / number; // Compute the average of the values
```

In the code line above, the executable statement and the comment are on the same line. The compiler will read the assignment but ignore the comment. The compiler has a way of generating the same code for this example and the one we gave below.

- **Block Comment**

This is the other type of comment that begins with the symbols /* and is only in effect when it ends with the symbols */. The block comment comes in handy when you want to comment in multi-lines. Consider the example below:

```
/* After the computation is completed, the result is printed. */
std::cout << result << ‘\n;
```

Formatting

Commenting on programs helps humans read and understand a code more, but the compiler ignores them. Another aspect of code sourcing that is largely irrelevant to the compiler is formatting. Imagine how hard it would be to read a text with no indention or space to separate paragraphs and words. The same applies to C++; source code formatting is equally important. Consider code 3.5 and code 3.6 and how formatting can be of use:

Code 3.5

```
#include <iostream>
int
main
(
)
{
int
a
;
a
=
5
;
std
;;
cout
<<
a
<<
‘\n’
;
}
```

Code 3.6

```
#include <iostream>
Int main () {int a;a=5;std;;cout<<a<<’n\;}
```

Both code 3.5 and code 3.6 are the same thing and are both valid C++ codes. However, most people would agree that Code 3.6, the formatted version of the original code in Code 3.5, is easier to read and understand more quickly. C++ is a kind of language that gives the programmer a vast freedom to format source code. A compiler reads a source code character by character (one symbol at a time) from left to right before going to the next line. Although spaces help to increase readability, spaces are not allowed in some places, for example, in variable names and reserved words; they must appear unbroken. Also, multi-symbol operators like << can’t be separated with space, amongst other examples.

Even if in C++, you’re not forced to use a particular kind of style while writing code, it helps to maintain a consistent style throughout the code you’re writing. Source codes that are not properly formatted takes a longer time to develop into a correct software because programmer’s mistake hides better in a poorly formatted code. There are tools you can use like the **pretty printer** to quickly format arbitrarily formatted C++ source code into a properly formatted one.

Errors and Warnings

Because beginners are inexperienced in programming, they tend to make a lot of mistakes while generating source codes. It could also be because they are not familiar with the programming language. That’s why they make a lot of mistakes. Whatever could be the reason for mistakes, programming error falls into one of three categories:

1. **Compile-Time Error**

When a developer misuses programming language, it results in a compile-time error. A syntax error is a common type of compile-time error. For example, the statement below is syntactically correct:

```
a = b + 2;
```

However, when we consider another example below, by replacing the assignment, and slightly modify the version, it becomes a syntax error.

```
b + 2 = a;
```

The statement above will report an error in the visual C++ compiler, among other things:

error C2106: ‘=’: left operand must be i-value

A compiler can also generate an error for a syntactically correct statement. Consider the example below,

```
a = b + 2
```

If the values of **a** and **b** have not been declared, it would cause an error. The visual C++ compiler would report

error C2065: ‘b’: undeclared identifier

2. **Run-Time Error**

The structure rule of a C++ language is not violated thanks to the compiler. The compiler can detect the malformed assignment statement and using of variables before declaration. Some types of violation of language can’t be detected at compile time. In a case like this, the program wouldn’t complete by run into an error known as the run-time error. We commonly associate this type of error in a program as “crashed.” Consider Code 3.7 for better understanding:

Code 3.7

```
// File dividedanger.cpp
#include <iostream>
int main () {
int dividend, divisor;
//Get two integer from the user
std::cout << “Please enter two integers to divide”;
std::cout >> divided >> divisor;
// divide them and report the result
std::cout << dividend << “/” << divisor << “ = “
<< dividend/divisor << ‘\n’;
}
```

Using the dividend/divisor expression can be potentially dangerous because if you instead of typing 20 and 5 type 20 and 0, the program would report and error and then terminates.

3. Logic Error

In Code 3.7 above, consider replacing the expression dividend/divisor with divisor/dividend. The compiler would run with no errors. This replacement works perfectly fine unless a value of zero is entered as the dividend. The answer it would compute with a zero dividend will not be correct. The only time when a correct answer would be printed is when the dividend is equal to the divisor. In other words, when the program has an error, and the compiler nor the run-time system can’t detect it, it’s known as logic error.

Also, errors that escape compiler detection are known as bugs. Because the compiler can’t detect the error, bugs are major issues for developers. This is particularly an issue because, in a complex program, the bugs are hard to find because they reveal themselves in certain situations, so they are difficult to reproduce while testing.

4. Compiler Warning

You may get a warning by your compiler which marks a violation of the rules of the C++ programming language. It is a notification to the developer that the code contains a construction that will potentially cause problems.

Consider Code 3.8 for more explanation:

Code 3.8

```
// unintialized.cpp
#include <iostream>
int main () }
int n;
std::cout << n << ‘\n’;
}
```

In the code example above, the programmer is attempting to print the value of a variable before giving it a known value. Trying to run this program on a compiler would display the following message on the Visual C++ compiler:

warning C4700: uninitialized local variable ‘n’ used

Arithmetic Examples

In C++, you can perform complex arithmetic expressions determined by an operator by operator basis. Take for example, in Code 3.9 below; we would be attempting to convert a temperature from degree Celsius to degree Fahrenheit

F = (C x 9/5) + 32

Code 3.9

```
#include <iostream>
int main () {
double degreesC, degreesF;
//Prompt user for temperature to convert
std::cout << “Enter the temperature in degrees C: “;
//Read in the user’s input
std::cin >> degreesF;
//Perfeorm the conversion
degreesF = (C x 9/5) +32;
// Report the result
std::cout << degreesF << ‘\n’;
}
```

Code 3.9 has documented comments that explains the purpose of each code. You can also make use of C++ to convers time by using modulus and integer division to split up the given number of seconds to hours, minutes and seconds. Consider code 3.10 for how it’s done:

Code 3.10

```
#include <iostream>
int main () {
int hours, minutes, seconds;
//First, compute the number of hours in the given number of seconds
Hours = seconds / 3600; // 3600 seconds = 1 hours
//Compute the remaining seconds after the hours are accounted for
seconds = seconds % 3600;
//Next, compute the number of minutes in the remaining number of
seconds
minutes = seconds / 60; //There are 60 seconds in a minute
//Compute the remaining seconds after the minutes are accounted for
seconds = seconds % 60;
//Report the results
std::cout << hours << “ hr, “ << minutes << “ min, “
<< seconds << “ sec\n”;
}
```

In the conversion of time code, if you had entered 10,000 the program would have printed 2 hours, 46 minutes, and 40 seconds.

Integers vs. Floating-Point Numbers

Using the floating-point numbers comes with a bunch of advantages than the integers. Using the double statement with the floating-point number have a much greater range of values than any other integer type. One of the advantages floating point has over integer number is can have fractional parts. Although integers have one big advantage over floating-point numbers because they are exact.

Bitwise Operators

Together with the common arithmetic operation, we introduced earlier, there are other special-purpose arithmetic operations. This special operation allows developers the freedom to manipulate and examine the individual bits making up data values. This whole special operator is otherwise known as bitwise operators. These operators consist of characters like <<, >>, ^, ~, |, and &. Generally, application developers do not need to make use of the bitwise operator very often. Bitwise has also been useful in bit manipulation, which is essential in so many systems programming tasks.

Algorithms

An algorithm is simply a finite sequence of steps where each step takes a finite length of time often used to solve problems and compute results. A computer program is an example of an algorithm, the same as a recipe to make lasagna. In whatever example, an algorithm is always pretty simple. Considering another example, if a and b are integer variable in a program. How can we interchange the values of the two variables is pretty easily? Well, we would want to give the value of a to b’s original value. Check out the statement below; it may seem reasonable:

```
a = b
b = a
```

But the issue here is with the section code coming after the first statement is executed. A and b have the same value of b’s original value. The second assignment does nothing in changing the values of a and b because it is superfluous. So, the solution here is that there is a need for a third variable that needs to remember what the original value is before it is reassigned. The correct way to write the code to swap is:

```
temp = x;
a = b
y = temp;
```

In the example above, there is an emphasis on the fact that algorithms have to be specified precisely. Even though informal notions on solving problems are valuable in the early stage of programing design, coding program requires a correct detail description of the solution.

Chapter 4

Conditional and Iterative Statements

In this chapter of this book, we shall be discussing what condition and iterative statements are. Condition execution is a construction of program statements that are optionally executed, but it depends on the context of the program's execution. But on the other hand, iteration repeats the execution of a sequence of code. They are very useful in solving a lot of programming problems. Iteration and conditional executions are basically the key components of a good algorithm construction.

Conditional Execution

Type of Bool

Just as an arithmetic expression evaluates to numerical values, the Boolean expression evaluates to **true** or **false**. Boolean expression may look as though it is very limited on the surface, but they are essential for building interesting and useful programs. C++ supports the use of non-numeric data types **bool,** which means Boolean. The word Boolean comes from a British mathematician George Boole. George Boole comes from a branch of discrete mathematic, also known as Boolean algebra dedicated to studying the properties and manipulation of logical expressions. Computing the numeric types, the **bool** type of data is very simple, and they can represent only two values, **true** or **false**. Consider the simple program below which demonstrate the use of Boolean variables

Code 4.1

```
#include <iostream>
int main () {
// Declare three Boolean variables
bool x = true, y = false
std::cout << “x = “ << x <<< “, y = “ << y << ‘\n’;
// assign a value to x
x = false
std::cout << “x = “ << x << “, y = “ << y << ‘\n’;
// Mix integers and Boolean
x = 0;
y = 1;
std::cout << “x “ << x << “, y = “ << y << ‘\n’;
// Assign Boolean values to an integer
int a = x, b = true;
std::cout << “x = “ << x << “, y = “ << y
<< “, a = “ << a << “, b = “ << b << ‘\n’;
// More mixing
x = 1725 // Warning issued
y = - 19 // Warning issued
std::cout << “x = “ << x << “, y = “ << y << ‘\n’;
}
```

So, as you can see from the running code 4.1 above, the Boolean values true and falls are represented as integers 0 and 1. T0 be more precise, zero represents the bool value of false while the non-zero value, either positive or negative, represents the bool value of true.

Boolean Expression

The simplest form of a Boolean expression is **true** or **false**, which are the Boolean literals. Boolean variables are also known as a Boolean expression. An expression that compares numeric expressions for inequalities and equalities is also known as a Boolean expression. The simplest form of Boolean expressions makes use of relational operators to compare two expressions. Consider the table below; it shows some common simple Boolean expression and their associated values.

Expression	Value
10 < 20	Always true
10 > - 20	Always false

x == 10	True only if x has the value of 10
x != y	True unless x and y have the same values

C++ allows you to use simple expressions for example the statement

x == 20;

This expression may look as though you're attempting to assign the values 20 to the variable x, but in truth, you're not. The = operator performs the assignment but on the other hand, the operator == checks for relational equality. But if you should make a mistake and use the == operator where you meant to use the = operator, then the Visual C++ compiler will issue a warning message as follows:

Warning C4553: '==': operator has no effect; did you intend '='?

This warning doesn't violate the rule of language, but it rather alerts you of a possible trouble spotted in the code.

The Simple IF Statement

The Boolean expressions are important in enabling a program to adapt its behavior at run time. A lot of practical and useful programs we see today are only possible because of the availability of the Boolean expression. The IF statement is a conditional statement that allows a program to be executable when certain conditions are met. Consider Code 4.2 below for a better understanding of how to use the IF statement.

Code 4.2

```
#include <iostream>
int main () {
int divided, divisor;
// Get two integers from the programmer
std::cout << "Please provide two integers to divide:";
std::cin >> divided >> divisor;
// If possible, divide them and report the result
if (divisor !=0)
std::cout << divided << "/" << divisor << " = "
<< divided/divisor << '\n';
}
```

The second **std::cout** may not always be executable. It depends on the values the user provided in the first **std::cout**. If the user provides two integers, let’s say 20 and 5, the program is executable. Still, if the user enters a zero as a second number, then the program will not print anything after the user enters the values.

Compound Statements

There are times when you may have to execute more than one statement, optionally based on a particular condition. Consider the source code in code 4.3 below on how you must use curly braces to group multiple statements into one compound statement

Code 4.3

```
#include <iostream>
int main () {
int divided, divisor, quotient;
// Request two integers from the programmer
std::cout << “Please enter the two integers to divivde:”;
std::cin >> dividend >> divisor;
// If possible, divide them and report the result
if (divisor ! = 0) {
quotient = dividend / divisor;
std::cout << dividend << “ divided by “ << divisor << “ is “
<< quotient << ‘\n’;
}
}
```

The printing and assignment statement are both a part of the **if** statement. Considering the true value of the Boolean expression divisor ! = 0 during a particular program run, either both statement be executed or neither of them will be executed. Compound statement consists of zero or more statements that are grouped in curly braces. We can say the curly braces define a block of statement. The programmer can also style up the code by always using curly braces to delimit the body of the **if** statement even if the body contains only a statement.

The IF/ELSE Statement

One undesirable aspect of C++ is that when a user enters a zero as a divisor, it always prints nothing. The Visual C++ compiler may issue a feedback that the indicated divisor can’t be used. But the **if** statement comes with an optional **else** clause. The **else** clause is executed only when the Boolean expression is false. Consider Code 4.4 to understand further how the if/else statement can be used.

Code 4.4

```
#include <iostream>
int main () {
int dividend, divisor;
// Request two integers from the programmer
std::cout << “Please provide two integers to divide:”;
std::cin >> dividend >> divisor;
// If possible, divide them and repost the result
if (divisor != 0)
std::cout << dividend << “/” << divisor << “ = “
<< dividend/divisor << ‘\n’;
else
std::cout << “Division by zero is not allowed\n”;
}
```

In Code 4.4 above, the **if** and **else** statement were used, and the program run will execute one of either. So, unlike the warning issue you get for entering a zero divisor, the else clause will alternate the body so that the program is executable. But when making use of the if/else statement in a statement, the following rule must apply:

- The reserved word **if** must begin the **if/else** statement
- The condition is a Boolean expression, which helps us to determine whether or not the running program will execute statement 1 or statement 2. As with the simple if statement, the condition must appear within the parentheses.
- The program will only execute statement 1 if it satisfies the conditions (true).

- To make the **if/else** statement more readable, indent statement 1 more than the **if** line. This section of the **if** statement is most often referred to as the body of the **if**.
- The reserved world else always begins the second part of the **if/else** statement.
- The program will only execute statement 2 if the condition is false.
- To make the **if/else** statement more readable, indent statement 2 more than the **else** line. This section of the **if/else** statement is sometimes called the body of the **else**.

Nested Conditionals

Within the body of the **if** or the **else** statement, there could be any C++ statements included other **if/else** statements, and in such cases, we call it nested conditionals. We can even nest an **if** statement to build arbitrarily complex control flow logic. Consider code 4.5 below to see how we determined if a number is between 0 and 10 inclusive.

Code 4.5

```
#include <iostream>
int main () {
int value;
std::cout << “Please provide a range of integer between 0 . . . 10: “;
std::cin >> value;
if (value >=0) // First check
if (value <= 10) // Second check
std::cout << “In range”;
std::cout << “Done\n”;
}
```

In code 4.5, the program checks the value >= 0 condition first. If the value is less than zero, the program would not be able to execute the second condition and so would not print **In range** but goes straight ahead and print the statement that follows the outer if statement which prints **Done**. Also, if the executing program finds a value to be greater than or equal to zero, it checks the second condition. If the second condition is met, it displays the **In range** message, but it is not met; it doesn’t display anything. Nevertheless, the program will print **Done** before it terminates.

Iteration

The While Statement

Consider the code 4.6 below; while counts five by printing a number on each output line.

Code 4.6

```
#include <iostream>
int main () {
std::cout << 1 << ‘\n’;
std::cout << 2 << ‘\n’;
std::cout << 3 << ‘\n’;
std::cout << 4 << ‘\n’;
std::cout << 5 << ‘\n’;
}
```

When you compile and run this code, it will display on the screen 1 through 5. But how would you count to 10,000? Would you have to copy, paste, and modify 10,000 printing statements? You could do that, but it would be impractical. Because counting is one of the most common activity computers do, there has to be a better way to count it. So, what we can really do is to print the value of a variable and call it **count**, then increment the variable (count++) and then repeat the process until the variable is large enough (count == 5 or perhaps count == 10000). Consider code 4.7 and see how we used the while statement to count to five.

Code 4.7

```
#include <iostream>
int main () {
int count = 1; // Initialize counter
while (count <= 5) {
std::cout << count << ‘\n’; //Display counter, then
count++; //Increment counter
}
}
```

Code 4.7 makes use of a while statement to display a variable that counts up to five. Unlike the approach taken in code 4.6, it is trivial to modify to count up to 10,000, just change the literal value 5 to 10000.

Nested Loops

Just like in the **if** statement, the **while** bodies can contain arbitrary C++ statements, including other **while** statements. A loop can also be nested within another loop. To best understand how the nested loop works, consider a program that prints out a multiplication table. While we were in elementary school, we were taught the products of integers up to 10 or even 12 in some cases. And the same can be applied in C++ to even a much larger number.

Abnormal Loop Termination

By default, the **while** statement executes, except its conditions become false. The abnormal loop that executes the program checks this condition only at the top of the loop. So, even if the Boolean expression makes up, the condition becomes false before the program completes executing all the statements within the body of the loop. The remaining statement in the loop's body has to complete. Otherwise, the loop can once again check its condition. So, the **while** statement cannot by itself exit its loop in the middle of its body. The abnormal loop termination can be divided into three sections:

1. **The Break Statement**

The C++ program provides a **break** statement that implements the middle-existing control logic. This statement causes the immediate exit from the body of the loop. Consider code 4.8 below as we illustrate the use of break

Code 4.8

```
#include <iostream>
int main () {
int input, sum = 0;
std::cout << “Enter number to sun, negative number ends list:”;
while (true) {
std::cin >> input;
if (input < 0)
break; // Exit loop immediately
sum += input;
}
std::cout << “Sum = “ << sum << ‘\n’;
}
```

The condition of the **while** in code 4.8 is a tautology. In other words, the condition is true and can never be false. When the statement reaches the while loop, it doesn’t provide a way out, so the if statement step in and provides a way out. In this scenario, the **break** statement is executed conditionally based on the value of the variable input. The break statement executes only when the programmer enters a negative number.

2. The goto Statement

As we already know, the break statement can exit the single loop in which it is located. So, a break statement can’t just jump completely out of the middle of a nested loop. But the goto statement, on the other hand, allows the statement which allows the program execution flow to jump to a specified location in the function. Consider code 4.9 and how we use a goto statement to jump out of the middle of a nested loop.

Code 4.9

```
#include <iostream>
int main () {
// Compute some products
int op1 = 2;
while (op1 < 100) {
int op2 = 2;
while (op2 < 100) {
if (op1 * op2 == 3731)
goto end;
std::cout << “Product is “ << (op1 * op2) << ‘\n’ ;
op2++;
}
op1++;
}
end:
std::cout << “The end” << ‘\n’;
}
```

When op1 * op2 is 3731, program flow it jumps to a specified label within the program. In this example, the label name might end, but the name is arbitrary. And like in variable names, the label names should be chosen to indicate their intended purpose. The label named end comes outside and after the nested **while** loops.

3. The Continue Statement

When a program’s execution encounters a **break** statement inside a loop, it skips the remaining body of the loop and then exits the loop. The **continue** statement is just like the **break** statement, except that the **continue** statement doesn’t necessarily exit the loop. In the continue statement, it skips the rest of the body of the loop and immediately checks the loop’s condition. If the loop’s condition remains true, then there would be an execution of the loops at the top of the loop. Consider Code 4.10 below to understand the continue statement in action better.

Code 4.10

```
#include <iostream>
int main () {
int input, sum = 0;
bool done = false;
while (!done) = false;
std::cout << Enter a positive integer (999 quits): “;
std::cin >> input;
if (input < 0) {
std::cout << “Negative value “ << input << “ ignored\n”;
continue; // skip rest of the body for this iteration
}
if (input != 999) {
std::cout << “Tallying “ << input << ‘\n’;
sum +=input;
}
else
done = (input == 99); // 999 entry exits loop
}
std::cout << “sum = “ << sum << ‘\n’;
}
```

Infinite Loops

An infinite loop is a type of loop with no exiting point. Once the program starts, it enters an infinite loop that it cannot escape. Some infinite loops are designed, for example, a long-running server application like a web server that needs to check for incoming connections continuously. This server application can perform this checking within a loop running indefinitely. Many times beginning programmers often create infinite loops by accident, which represent logic errors in their programs. An intentional infinite loop should look obvious. For example:

```
While (true) {
/* Do something forever . . . */
}
```

In Boolean literal, true is always true, and so a loop’s condition can’t be false. One way to exit a loop like this is with a **break** statement, **exit** call, or **return** statement embedded within the body. It’s quite easy to write an intentional infinite loop.

Chapter 5

Using, Writing and Managing Functions and Data

In this section of this book, you'd learn how to use, write, and manage functions. This is a very wide section of C++, but we would try out best to cover as much part of it as we can. So, without much ado, let's get right into it.

Introduction to Using Functions

In mathematics, a function is want is used to compute results from a given value. Take for examples, the function f(x) = 3x + 6, we can compute that f(2) = 12 and f(0) = 6. In C++, a function works like a mathematical function. In C++ functions are named sequence of code that performs a certain task. And a program can consist of a collection of function. For example, functions with mathematical square root function are named **sqrt**. Using the square root function accepts one numerical value and then produces a **double** value as a result. Take for example the square root of 25 is 5, so when presented with sqrt 25, the responds will be 4.0. Code 5.1 better explains:

Code 5.1

```
#include <iostream>
#include <cmath>
int main () {
double input;
// Get the value from the programmer
std::cout << Enter number: ";
std::cin >> input;
// Compute the square root
double root =  sqrt (input);
```

```
        // Report result
        std::cout << “Square root of “ << input << “ = “ << root << ‘\n’;
        }
```

Standard Mathematic Function

The **cmath** function has a lot of functionality in a scientific calculator. The functions in the cmath lab are ideal for solving scientific functions. The table below shows a few functions from the cmath library.

Math module	Function
double sqrt (double x)	Computes the square root of a number: sqrt (x) = $\sqrt{x}$
double exp (double x)	Computes e raised to the power : exp (x) = e^x
double log (double x)	Computes the natural logarithm of a number: log (x) = $\log_e x$ = lnx
double log10 (double x)	Computes the common logarithm of a number: log(x) = $\log_{10} x$
double cos (double)	Computes the cosine of a value specifies in radians: cos(x) = cosx; other trigonometric function including the since, tangent, hyperbolic cosine, hyperbolic tangent, hyperbolic sine, arc tangent, arc cosine, and arc cosine.
double pow (double x, double y)	Raises one number to the power of another: pow (x, y) = x^y
double fabs (double x)	Computes the absolute value of a number: fabs (x) = $\|x\|$

Maximum and Minimum

C++ also provides functions that programmers can use to determine the maximum and minimum of two numbers. Code 5.2 better explains the way C++ helps in determining min and max functions.

Code 5.2

```
#include <iostream>
#include <algorithm>
int main () {
int value1, value2;
std::cout << “Please enter the two values for the integer: “;
std::cout >> value1 >> value2;
std::cout << “max = “ << std::max(value1, value2)
<< “, min = “ << std::min(value1, value2) << ‘\n’;
}
```

The main thing to note about using the standard max and min function in a program is to include the <algorithm> header.

Clock Function

The clock function from the **<ctime>** library can be used to request from the operating system the amount of time a program you executed has been running. This unit, returned by the call to **clock (),** is dependent on the system, but it can be converted to seconds with the constant **CLOCKS_PER_SEC**, which is also defined in the **<ctime>** library. In the Visual C++, **CLOCKS_PER_SEC** constant is 1,000, meaning the calling clock () returns the number of milliseconds that the program has been running. When you make use of two calls to the clock function, you can measure elapsed time. See code 5.3, how we measured how long it takes for a user to enter a character from the keyboard.

Code 5.3

```
#include <iostream>
#include <ctime>
int main () {
char letter;
std::cout << “Enter a character: “;
clock_t seconds = clock () ; // Record the starting of the time
std::cin >> letter;
clock_t other = clock () ; // Record the ending of the time
std::cout << static_cast<double>(other –
seconds)/CLOCKS_PER_SEC
<< “ seconds\n”;
}
```

The type **clock_t** is a function that defined in the **<ctime>** header. **Clock_t** is similar to the **unsigned long**, and you can use it to perform arithmetic functions on **clock_t** variables and values.

Character Function

In the C library, there are some character functions that are useful to C++ programming. Consider Code 5.4 and how we converted lowercase letters to uppercase letters.

Code 5.4

```
#include <iostream>
#include <cctype>
int main () {
for (char lower = ‘a’; lower <= ‘z’; lower++) {
char upper = toupper (lower);
std::cout << lower << “ => “ << upper << ‘\n’;
}
}
```

In code 5.4 above the first lines prints a as A, b as B, c as C, d as D all the way to z as Z. interestingly, the function in code 5.4 returns an **int** and not a **char**. At the enhanced warning level 4 for visual C++, a cast is required to assign the result to the variable upper.

Random Numbers

Some applications require behavior that may appear as though it is random. Random numbers are useful, especially in simulation and games. Take, for instance, a lot of board games that make use of die or dice to determine how many places a player can move. A die or dice are mostly used in games of chances. A software that adapts the use of dice would need to find a way to simulate the random roll of a dice.

All algorithms random number generators can be used to produce pseudorandom numbers. A pseudorandom number generator has a period based on the nature of the algorithm being used. If the generator is used long enough, the pattern of the number it produced will start to repeat itself. So, a sequence of true random numbers would not contain such a repeating subsequence. But the good news is that most pseudorandom number generators have a period large enough for most applications.

C++ programmer s make use of two standard C functions **srand** and **rand** to generate pseudorandom numbers. **srand** can be used to establish the first values in the sequence of pseudorandom integer values. And each call of **rand** returns the next value in the sequence of pseudorandom value. Code 5.5 shows how to generate a sequence of 100 pseudorandom number:

Code 5.5

```
#include <iostream>
#include <cstdlib>
int main () {
srand (23) ;
for (int i = 0; i < 100; i++) {
int r = rand () ;
std::cout << r << " " ;
}
std::cout << ''\n';
}
```

The number that will be printed by the program will appear to be random. The algorithm is given a seed value to start and a formula to produce the next value. The seed value is what determines the sequence of the number generated. In other words, identical seed values generate identical sequences. If you run the program again, the same sequence is then displayed because the same seed value is used. To allow the program to run to display different sequences, the seed value must be different for different runs.

Writing Functions

When writing a program, it gets more complex as you proceed, and it would help if programmers structure their program in a way as to manage their complexity effectively. So far, we've been writing our programs within one function - **main**. As the number in a statement within a function begins to increase, the function can become unwieldy. But the code that is within such a function that does the work by itself is called the monolithic code. Monolithic codes that are complex and long are undesirable for many reasons, especially it being difficult to write correctly, difficult to debug, and difficult to extend. So, if the function’s purpose is generally enough and you can write the function well, then we would be able to rescue the function in other programs as well.

Function Basics

Remember the handwritten square root code we explained in our former chapter. Now let’s use that as an example to compare the behavior of custom square_root function with the sqrt library function. Consider code 5.6 and see how we achieve that.

Code 5.6

```
#include <iostream>
#include <iomanip>
#include <cmath>
// Compute an approximation of the square root of x
double square_root(double x) {
double diff;
// Compute a provisional square root
double root = 1.0;
do { // Loop until the provisional root is close enough to the actual root
root = (root + x/root) / 2.0;
//std::cout << "root is " << root << '\n';
// How bad is the approximation?
diff = root * root - x;
} while (diff > 0.0001 || diff < -0.0001);
return root;
}
int main() {
// Compare the two ways of computing the square root
for (double d = 1.0; d <= 10.0; d += 0.5)
std::cout << std::setw(7) << square_root(d) << " : " << sqrt(d) << '\n';
}
```

The output shows only a small difference in the results. Basically, there are two aspects of the C++ function:

Function Definition

Defining a function specifies the function's return type and parameter types. It provides the code that determines the function's behavior

Function Invocation

When a programmer uses a function via a function invocation, the main function invokes both the **square_root** function and the **sqrt** function. Every function has exactly one definition, and it may have many invocations.

Furthermore, a function consists of up to four major parts:

Name

Every function in the C++ language has a name. The name of the function is an identifier. The same thing applies to variables. Variables have names, and the names were chosen for a function that should accurately portray its intended purpose and functionality.

Type

Every function also has a return type. If the function returns a value to its caller, then its type corresponds to the type of value it returns. The special type **void** signifies that the function doesn't return a value.

Parameters

A type of parameter must also specify every function that it accepts from callers. The parameter appears in a parenthesized comma-separated list like a function prototype. Unlike function prototype.

Body

Every function in the C++ language also has a body enclosed by curly braces. The body contains the code to be executed when the function is invoked.

Using Functions

The general form in which we ought to define function is

```
    Type name ( parameterlist ) {
```

```
Body
}
```

The type of function tells us the type of value the function can return. Many times, a function will perform a calculation, and the result of the calculation has to be communicated back to the place where the function was invoked. The special type **void** can be used to indicate that the function doesn't return a value. If the name of the function is an identifier, the function's name should indicate the purpose of the function. And if the parameterlist is a comma separating the list of pair of the forms

```
type name
```

in which type is a C++ type, and name is an identifier representing a parameter.

The caller of the function communicates information into the function via parameter. The parameter in the parameter list of a function definition is called formal parameters. A parameter can also be known as an argument. Even though the parameter list may be empty, an empty parameter list indicates there are no information passed into the function by the caller. And if the body is in a sequence of statements, it has to be enclosed within curly braces. The body enclosed in curly braces defined the actions that the function is to perform. These statements could include variable declarations and variables declared within the body that are local to the function.

Commenting Functions

Like we've always said, it is a good practice to always comment on a function as it provides information that aids programmers who may need to use or extend the function. The essential information includes:

1. The purpose of the function
2. The role of each parameter
3. The nature of the return value

There are also other information you may often require in a commercial environment:

1. Author of the function
2. Date that the function's implementation was last modified
3. References

The following code 5.7 is an example of a fragment showing the beginning of a well-comented function definition:

Code 5.7

```
/* * distance(a1, b1, a2, b2)
* Computes the distance between two geometric points
* a1 is the a coordinate of the first point
* b1 is the b coordinate of the first point
* a2 is the a coordinate of the second point
* b2 is the b coordinate of the second point
* Returns the distance between (a1,b1) and (a2,b2)
* Author: Author's name
* Last modified: 2020-16-04
* Adapted from a formula published at
* http://www.xxxxxxxx.com
*/
double distance(double a1, double b1, double a2, double b2) {
```

Managing Functions and Data

It's also important you understand some additional aspects of functions in the C++ language. Recursion is a key concept in computer science today.

Global Variables

All variables to this point have local to blocks or local to functions within the bodies of iterative or conditional statements. The local variable has some really interesting properties like:

1. Local variables can only occupy a memory only when their variable is in scope. When the program execution leaves the scope of a local variable, it frees up the memory for that variable. The memory that has been freed up is then available for use again by the local variable in other functions during their

invocations.

2. It is possible to make use of the same variable name in different functions without any conflict. The compiler drives all of its information about a local variable used within a function by the declaration of its variable in the function. The compiler doesn’t take note of the declaration of a local variable in another function. In other words, there is no way a local variable in the definition of another function can interfere with a local variable in another function.

Local variables are transitory, and so their values are lost in between function invocations. Sometimes, it is better to have a variable that lives as long as the program is running until the main function completes.

Static Variables

In the computer’s memory, the spaces for local variables and function parameter is allocated at run time when the function begins to execute. After the function has executed and returned, the memory used for the function’s local variables and parameters are freed up for other purposes. But if you don’t call a function, the variable’s local variables and parameters will never occupy the computer’s memory.

So, because of the transitory nature of local functions, a function can’t by itself retain any information between calls. So, C++ language provides a way in which a variable local to a function can be retained in between calls. Code 5.8 allows us to show you that by declaring a local variable static, it allows it to remain in the computer’s memory for the duration of the program’s execution.

Code 5.8

```
#include <iostream>
#include <iomanip>
//Count
//Keeps track of a count.
//Returns the current count
int count() {
// cnt's value is retained between calls because it
// is declared static
static int cnt = 0;
return ++cnt; // Increment and return current count
}
int main() {
// Count to ten
for (int i = 0; i < 10; i++)
std::cout << count() << ' ';
std::cout << '\n';
}
```

Overloaded Function

In the C++ language, a program can have more than one function having the same name. But when two of more functions within a program have the same names, the function is said to be overloaded. Function within a program must be different somehow, or else the compiler would not know how to associate a call with a particular function definition. Mainly, a compiler identifies a function with more than its name; its signature can uniquely identify a function. The signature of a function is distinguished by its name and parameter list. In the parameter list, it is only the types of formal parameters that are important and not their names. If the parameter type doesn't match exactly, both the position and number, then the function signature are different.

Default Arguments

We can also define a function that accepts a wide range of number of parameters. Consider code 5.9 below that specifies a function that counts down:

Code 5.9

```
#include <iostream>
// Prints a count down from n to zero.
// The default starting value is 10.
void countdown(int n=10) {
while (n >= 0) // Count down from n to zero
std::cout << n-- << '\n';
}
int main() {
countdown(10);
std::cout << "----------" << '\n';
countdown();
}
```

Recursion

The function factorial is widely used in combinatorial analysis (counting theory in mathematics), probability theory, and statistics. The factorial of **n** is most times expressed as n!. The factorial for nonnegative integers can be expressed as:

$$n! = n \,.\, (n - 1) \,.\, (n - 2) \,.\, (n - 3) \,.\,.\,.\, 2 \,.\, 1$$

Mathematically, the definition of factorial is recursive because the ! function is being defined, but ! is also used in the definition. In C++, a function can be defined recursively as well. Consider code 5.10, which includes a factorial function that exactly models the mathematical definition.

Code 5.10

```
#include <iostream>
// Factorial (n)
// Computes n!
// Return the factorial of n;
int factorial(int n) {
if (n == 0)
return 1;
else
return n * factorial(n - 1);
}
int main() {
// Try out the factorial function
std::cout << " 0! = " << factorial(0) << '\n';
std::cout << " 1! = " << factorial(1) << '\n';
std::cout << " 6! = " << factorial(6) << '\n';
std::cout << "10! = " << factorial(10) << '\n';
}
```

So, a simple correct recursive function definition is based on four key concepts:

1. The function must not be able to call itself within a definition optionally. This is also known as the base case.

2. The function must be able to call itself within its definition optionally. This is also known as the recursive case.

3. Any invocation that doesn’t correspond to the base case must be able to call itself with parameters that move the execution closer to the base case. The functions recursive execution must cover to the base case

4. And lastly, some sort of conditional execution like the use of **if/else** statement select between the recursive case and the base case based on one or more parameters passed to the function.

Chapter 6

Sequences

So far, so good, we’ve been making use of variables that can only assume one value at a time. And as we can see, it is possible to use individual variables to form something useful and interesting in a program. Nevertheless, variables have a limitation, like their ability to only represent one value at a time. Consider Code 6.1 below, which shows the average of five numbers entered by a programmer.

Code 6.1

```
#include <iostream>
int main() {
double a1, a2, a3, a4, a5;
std::cout << "Please enter five numbers: "; //
Allow the user to enter in the five values.
std::cin >> a1 >> a2 >> a3 >> a4 >> a5;
std::cout << "The average of " << a1 << ", " << a2 << ", "
<< a3 << ", " << a4 << ", " << a5 << " is "
<< (a1 + a2 + a3 + a4 + a5)/5 << '\n';
}
```

The program above, when compiled, conveniently displays the values the programmer entered and then computes the average. If the value to average must increase from five to let’s say twenty-five, then we must introduce twenty more additional variables, and overall the length of the program would increase. But if we were to consider averaging up to a thousand number, with this approach, it would be impractical. Consider Code 6.2, which shows an alternative way to average larger numbers easily.

Code 6.2

```
#include <iostream>
```

```
int main () {
double sum = 0.0, num;
const int NUMBER_OF_ENTRIES = 5;
std::cout << "Please enter " << NUMBER_OF_ENTRIES << "
numbers: ";
for (int i = 0; i < NUMBER_OF_ENTRIES; i++) {
std::cin >> num;
sum += num; }
std::cout << "The average of " << NUMBER_OF_ENTRIES << "
values is "
<< sum/NUMBER_OF_ENTRIES << '\n';
}
```

Code 6.2 can easily be modified to average up to 25 values than Code 6.1 by changing the constant NUMBER_OF_ENTRIES. Moreover, Code 6.2 averaging up to one thousand numbers is not difficult. However, unlike Code 6.1, Code 6.2 does not display the numbers enters, so it may be necessary to retain all the values you entered into the code for several reasons. In more situations, we must retain the values we entered into the code in case of future reference. We need to combine the advantages of Code 6.1 and Code 6.2. Specifically, we want to be able to retain every individual value and also avoid defining variables to store all the individual values. These may seem like a contradictory requirement, but C++ provides several standard data structure that provides both of these advantages.

In this chapter, our focus will be to examine the common sequence types available in C++ vectors and arrays. Vectors and arrays are sequence types because a sequence simply means ordered elements. A non-empty sequence comes with the following properties:

- All non-empty sequences have a unique first element
- All non-empty sequences have a unique last element
- All elements in a non-empty sequence apart from the first element have a unique predecessor element
- All elements in a non-empty sequence apart from the last element have a unique successor element

We refer to this as linear ordering. When using the linear ordering, you can start from the first element and then repeatedly visit successor elements until you reach the last element. There is never any ambiguity when it comes to which element is next in a sequence. The data structure we would be examining in this chapter would be sequence types like the std::vector, std::arrays, and primitive arrays.

Vectors

A vector in C++ is an object which helps in managing a block of memory for holding multiple values simultaneously. In other words, a vector represents a collection of values. Vectors are usually names, and we can access the value it contains through their position within the block of memory managed by the vector. Vectors can also store a sequence of values where the values must be of the same type. A collection of values, all of the same type, is said to be homogenous.

Declaring and Using Vectors

If you want to declare a vector object in the C++ program, then you must add the processor directive:

```
#include <vector>
```

The vector type is a part of the standard (std) namespace, so its full name is std::vector, which is similar to the full name of cout is std::count. If you were to include the directive

```
using std::vector;
```

in a source file, then make use of the shorter name of vector inside the code. We can also declare a vector object capable of holding integers as:

```
std::vector<int> vec_x;
```

In the statement above, the vec_x initially hold ten integers. All the ten elements are zero by default. However, the vector’s size appears within parentheses following the vector’s name. We can also declare a vector with a particular initial size as follows:

```
std::vector<int> vec_y(10);
```

Note that the first number within the parentheses following the vector's name indicates the number of elements, while the second argument specifies the initial value of all the elements. We may declare a vector and specify every element separately:

```
std::vector<int> vec_z{5, 10, 15, 20, 25};
```

Note that the elements in the statement above appears within curly braces and not parentheses. Elements within the curly braces are the vector initializer list.

Traversing a Vector

Traversing a vector simply means the action of moving through a vector visiting each element. **for** loops are ideal for traversing a vector. For example, if **x** is an integer containing ten elements, the following loop prints each element in:

```
for (int i = 0; i < 10; i++)
std::cout << x[i] << ‘\n’;
```

In a loop control variable, **i**, steps through each of the valid index of vector **x**. The value of the variable **i** starts from 0 and ends at 9, which is the last valid position in vector x. The following loop prints in vector **x** in a reverse order:

```
for (int i = 9; i >= 0; i--)
std::cout << x[i] << '\n';
```

And if you want to produce a vector named set which contains the integer sequence 0, 1, 2, 3, 4, …, 999, then the following code:

```
std::vector<int> set(1000);
for (int i = 0; i < 1000; i++)
set[i] = i;
```

Now that we understand the basics of traversing a loop, we now have all we need to build a program that flexibility averages numbers and at the same time retains all the value entered. Consider Code 6.3 which makes use of a vector and a loop to achieve the generality and ability to retain all input for later redisplay.

Code 6.3

```
#include <iostream>
#include <vector>
int main() {
double sum = 0.0;
const int NUMBER_OF_ENTRIES = 5;
std::vector<double> numbers(NUMBER_OF_ENTRIES);
std::cout << "Please enter " << NUMBER_OF_ENTRIES << "
numbers: ";
// Allow the user to enter in the values.
for (int i = 0; i < NUMBER_OF_ENTRIES; i++) {
std::cin >> numbers[i];
sum += numbers[i];
}
std::cout << "The average of ";
for (int i = 0; i < NUMBER_OF_ENTRIES - 1; i++)
std::cout << numbers[i] << ", ";
// No comma following last element
std::cout << numbers[NUMBER_OF_ENTRIES - 1] << " is "
<< sum/NUMBER_OF_ENTRIES << '\n';
}
```

Vector Methods

Since a vector is an object, and objects are different from the simple types like **int**, **bool**, and **double** in so many ways. A lot of objects have access to special functions known as methods. As a programmer in C++, you can refer to this method as member functions. A method is a function that is associated with a class of objects. A method invocation consists of a slightly different syntax than the function invocation. Consider **obj,** for example; if it is an object that supports a method named **f**, which cannot accept parameters, then we can invoke the **f** on behalf of the **obj** with the statement:

```
obj.f();
```

The dot operator in the statement connects the object with a method to invoke. Apart from this special invocation syntax, this method works pretty well in global functions. A method can also accept parameters and also return a value. Vectors can also support several methods, but we will be focusing on seven of them currently:

- push_back: This method inserts a new element onto the back of a vector
- pop_back: This method removes the last element from a vector
- operator []: This method provides access to the value stored at a given index within the vector
- at This method provides bounds-checking access to the value stored at a given position within the vector
- size: This method returns the number of values currently stored in the vector
- empty: This method returns true if the vector contains no elements, and it returns false if the vector contains at least one element.
- clear: This method makes the vector empty.

Vectors and Functions

It is possible to note that a function can accept a vector. Consider Code 6.4, which chooses how a function can accept a vector.

Code 6.4

```
#include <iostream>
#include <vector>
//print(x)
void print(std::vector<int> x) {
for (int elem : x)
std::cout << elem << " ";
std::cout << '\n';
}
//sum(x)
int sum(std::vector<int> x) {
int result = 0;
for (int elem : x)
result += elem;
return result;
}
int main() {
std::vector<int> list{ 5, 10, 15, 20, };
// Print the contents of the vector
print(list);
// Compute and display sum
std::cout << sum(list) << '\n';
// Zero out all the elements of list
int n = list.size();
for (int i = 0; i < n; i++)
list[i] = 0;
// Reprint the contents of the vector
print(list);
// Compute and display sum
std::cout << sum(list) << '\n';
}
```

The print definition:

```
void print(std::vector<int> x) {
```

shows that a former vector parameter can be declared just like a non-vector parameter. In a case like this, the print function makes use of pass by values so that during the program's execution, an invocation of print will **copy** the data into the actual parameter (**list**) to the formal parameter (**x**).

Multidimensional Vectors

So far, so good, we have been considering one-dimensional vector – simple sequences of values. But C++ also supports higher dimensional vectors like a two-dimensional vector, which is best visualized as a table with columns and rows. Consider the statement below:

```
std::vector<std::vector<int>> x(2, std::vector<int>(3));
```

This statement effectively declares **x** as a two-dimensional vector of integers. The statement literally creates a vector with two elements where each element is itself a vector that contains three integers. Take note that the type of **x** is a vector of integers. A two-dimensional vector can sometimes be called a matrix, whereby, in this case, the declaration specifies that two-dimensional vector **x** contains three columns and two rows.

Arrays

C++ programming language is object-oriented, and a vector is a perfect example of a software object. The C++ began as an extension of the C programming language, but C doesn't directly support object-oriented programming. In other words, C doesn't have vectors for sequence type representation. C programming language makes use of a more primitive construct called arrays. An array is a variable used to refer to a block of memory that is like a vector but can hold up multiple values at the same time. An array usually has a name, and one can access the value it contains through the position within the block of memory designed for the array. And just like in vector, the elements within an array has to be of the same type. Array can also be made of global or local variables. Arrays are also built into the core language of both C++ and C. Meaning you don't have to add any **#include** directive to make use of an array in a program.

Static Arrays

Basically, an array has two varieties, dynamic and static. When declaring an array, you must supply the size of a static array. For example, consider the statement below:

```
// list is an array of 20 integers
int list[20];
```

This statement makes use of the list declaration for an array of 20 numbers. The value in the square brackets is what specifies the number of elements in the array, and the size is fixed for the life of the array. The value in the square brackets must be a constant value that will be determined at compile time.

Pointers and Arrays

The name of an array used in C++ source code references a location in memory, which is the elements at index 0 in the array. So, an array name can be similar to a constant pointer. For this reason, we can treat an array identifier like a pointer. In the same manner, we can direct a pointer to point to an array and then treat the pointer like an array. Consider Code 6.5 on how to use a pointer to traverse an array.

Code 6.5

```
#include <iostream>
int main() {
int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
*p;
p = &x[0]; // p points to first element of array x
// Print out the contents of the array
for (int i = 0; i < 10; i++) {
std::cout << *p << ' '; // Print the element p points to
p++; // Increment p so it points to the next element
}
std::cout << '\n';
}
```

From Code 6.5, the statement:

```
p = &x[0];
```

sets **p** to point to the first element of array **x**. a shorter way to accomplish the same thing is to use the statement:

```
p = x;
```

because **x** is itself a reference to the array’s location in the memory. The assignment statement demonstrate clearly the association between pointer variables and array variables. Note that the opposite assignment (x = p) is not possible.

Dynamic Arrays

Developers do not need to worry about managing memory used by static arrays. Your compiler and run-time environmental automatically ensure that the array has about enough space to hold all its elements. Spaces held up by local arrays are freed up automatically when the local array is out of the scope of its declaration. The significant limitation of static array is the size, which is determined at compile time. Although the programmer can change the size of the array in the source code before recompiling it, but once the program is compiled into an executable program, any static array’s size is fixed. An approach that can help implement a flexibly-sized array defines the array to hold as many items as it conceivably will ever need in Code 6.6.

Code 6.6

```
#include <iostream>
// Maximum number of expected values is 1,000,000
const int MAX_NUMBER_OF_ENTRIES = 1000000;
double numbers[MAX_NUMBER_OF_ENTRIES];
int main() {
double sum = 0.0;
int size; // Actual number of entries
// Get effective size of the array
std::cout << "Please enter number of values to process: ";
std::cin >> size;
if (size > 0) { // Nothing to do with no entries
std::cout << "Please enter " << size << " numbers: ";
// Allow the user to enter in the values.
for (int i = 0; i < size; i++) {
std::cin >> numbers[i];
sum += numbers[i];
}
std::cout << "The average of ";
for (int i = 0; i < size - 1; i++)
std::cout << numbers[i] << ", ";
// No comma following last element
std::cout << numbers[size - 1] << " is " <<
sum/size << '\n';
}
}
```

Copying an Array

It is important to take note that in a C++ source code, a static array viable behaves similarly to a constant pointer. It may seem plausible at first, to make a copy of an array, as follows in Code 6.7:

Code 6.7

```
int x[5], y[5]; // Declare two arrays
for (int i = 0; i < 5; i++) // Populate one of them
x[i] = i; // x is filled with increasing values
y = x; // Make a copy of array x?
```

Since **y** behaves like a constant pointer, we can't reassign it. Meaning **y** can't appear on the left side of the assignment operator all by itself. Wherever **y** points, it must continue to point there during its lifetime. The source code in Code 6.7 will not compile. Even when we make use of a dynamic array, the assignment simply makes the two-pointer point to the same block of memory, which does not have the same effect as a vector assignment. Code 6.8 shows the proper way to make a copy of array x:

Code 6.8

```
int x[5], *y; // Declare two arrays, one dynamic
for (int i = 0; i < 5; i++) // Populate one of them
x[i] = i; // x is filled with increasing values
// Really make a copy of array x
y = new int[10]; // Allocate y for (int i = 0; i < 5; i++)
y[i] = x[i];
```

Multidimensional Arrays

Just like multidimensional vectors, multidimensional arrays is also supported by C++ language. Consider the following statement:

```
int x [2] [4];
```

This statement declares that **x** is a two dimensional array of integers. In this scenario, the declaration specifies that array **x** contains four columns and two rows. Using a syntax similar to that of vector, we could declare and create a two dimensional array above as:

```
int x [2] [4] = { { 5, 23, 15, 9},
                  { 12, 39, -8, 19} };
```

In multidimensional arrays, we can omit the ROW size in the parameter declaration, but as for the second set of square brackets, must contains a constant integral expression. Declaring parameters are quite complicated, and as we simplified it for vector, the same applies.

C Strings

In the C++ programming language, the C String is a sequence of characters. Both the C and C++ programming languages, make use of the C Strings as an array of **char**. The C++ language also supports string objects. The only time you use the char array is when writing the C language. We can also make use of the C string to refer to an array of characters as in the C language. A C string is an array of characters. A C string literal is a sequence of characters enclosed in quotation marks as in the statement below:

```
std::cout << "Hello!\n";
```

All properly used C strings are all null-terminated. In other words, the last character in the array is ASCH zero, which means that C++ represents the character literal **'\0'**. Since stings are true arrays, then we must take caution when using the sting variables, especially:

- When enough space has to be reserved for several characters in the string, including the null terminating character.
- When the array of characters must be properly null-terminated.

The following code fragment is acceptable and can be used safely:

```
char *word = "Hello!";
std::cout << word << '\n'
```

In the code fragrance above, the variable **word** was declared to be a pointer to a character, and it is initialized to point to a string literal. Furthermore, the code fragrance below is less safe to use:

```
char word[479];
std::cin >> word;
```

Chapter 7

Sorting and Searching

In our previous chapter, we introduced the fundamentals of making and using vectors and arrays. In this chapter, our focus will be to explore some common algorithms for ordering elements within a vector and also to locate elements using their value and not their index.

Sorting

In this chapter, we would make use of the genetic term sequence to refer to either an array or a vector. Sorting, in this case, arranges the elements in a sequence to form a particular order, which is a common activity. For instance, you could arrange a sequence in ascending order (from the smallest to the largest). So also, a sequence of words (strings) may be arranged lexicographically (alphabetic order). There are so many sorting algorithms, and some of them can perform a lot better than others. Let us consider one sorting algorithm that is very easy to implement.

The selection sort algorithm is very easy to implement, as it performs acceptably for smaller sequences. If **X** is a sequence, and ***i*** and ***j*** represents indices in the sequence, then selection sort works as follows:

1. Set ***i*** = 0
2. Examine every element in **X** [***j***], where ***j*** > ***i***. If peradventure, any of these elements is less than **A** [***i***], then exchange **X** [***i***] with the smallest of these elements. (But ensure that all the elements after positioning ***i*** are greater than or equal to **X** [***i***]).
3. Lastly, if ***i*** is less than the length of **X**, then you must increase ***i*** by 1 and proceed to Step 2.

If in step 3 the conditions are not met, then the algorithm would terminate with a sorted sequence. The command line “goto Step 2” in Step 3 is a loop. Let’s begin to translate the above description in C++ as follows.

```
// n is X's length
for (int i = 0; i < n - 1; i++) {
// Examine all the elements // X[j], where j > i.
// If any of these X[j] is less than X[i],
// then exchange X[i] with the smallest of these elements.
}
```

From the direction in Step 2, let’s begin with “Examine all the elements X[*j*], where *j* > *I*” also requires a loop. Let’s continue to refine our implementation by using:

```
// n is X's length
for (int i = 0; i < n - 1; i++) {
for (int j = i + 1; j < n; j++) {
// Examine all the elements
// X[j], where j > i.
}
// If any X[j] is less than X[i],
// then exchange X[i] with the smallest of these elements.
}
```

For us to determine whether the elements are less than X[i] then we must introduce a new variable named **small**. The purpose of introducing the **small** variable is to keep track of the position of all found small element. So, we set **small** equal to **i** which is the initial because we want to locate any element that is less than the element found in the position of **i**.

```
// n is X's length
for (int i = 0; i < n - 1; i++) {
// small is the position of the smallest value we've seen
// so far; we use it to find the smallest value less than X[i]
int small = i; for (int j = i + 1; j < n; j++) {
if (X[j] < X[small])
small = j; // Found a smaller element, update small
}
// If small changed, we found an element smaller than X[i]
if (small != i)
// exchange X[small] and X[i]
}
```

Flexible Sorting

Flexible sorting can be used to change the behavior of a sorting function so that it arranges the elements in a descending order rather than ascending order. Flexible sorting is actually pretty straightforward and easy to modify. For example, consider the line below

```
if (x[j] < x[small])
```

We can rewrite the statement using flexible sorting as:

```
if (x[j] > x[small])
```

And if we do like, we can change the sort in a way that the sorted elements are in an ascending order apart from all the even numbers in the vector appearing before the odd numbers. Although it would take a little bit more effort, but it is possible to do so. Here's an intriguing question, ask yourself how can we rewrite the **selection_sort** function in a way that by passing additional parameters, it would still sort the vector in any way we would like it? Well, this is possible, and we can make our sort function more flexible through higher-order function. Consider Code 7.1 for a more precise arrangement of elements in a vector two different ways using the same **selection_sort** function.

Code 7.1

```
#include <iostream>
#include <vector>
/*
* less_than(x, y)
* Returns true if x < y; otherwise, returns
* false.
*/
bool less_than(int x, int y) {
return x < y;
}
/*
* greater_than(x, y)
* Returns true if x > y; otherwise, returns
* false.
*/
bool greater_than(int x, int y) {
return x > y;
}
/*
* selection_sort(x, compare)
* Arranges the elements of x in an order determined
* by the compare function.
* x is a vector of integers.
* compare is a function that compares the ordering of
* two integers.
*/
void selection_sort(std::vector<int>& x, bool (*compare)(int, int)) {
int n = x.size();
for (int i = 0; i < n - 1; i++) {
// Note: i,small, and j represent positions within x
// x[i], x[small], and x[j] represents the elements at
// those positions.
// small is the position of the smallest value we've seen
// so far; we use it to find the smallest value less
// than x[i]
int small = i;
// See if x smaller value can be found later in the vector
for (int j = i + 1; j < n; j++)
if (compare(x[j], x[small]))
small = j; // Found x smaller value
// Swap x[i] and x[small], if x smaller value was found
if (i != small)
std::swap(x[i], x[small]); // Uses std::swap
}
}
/*
```

```
* print
* Prints the contents of an integer vector
* x is the vector to print.
* x is not modified.
*/
void print(const std::vector<int>& x) {
int n = x.size();
std::cout << '{';
if (n > 0) {
std::cout << x[0]; // Print the first element
for (int i = 1; i < n; i++)
std::cout << ',' << x[i]; // Print the rest
}
std::cout << '}';
}
int main() {
std::vector<int> list{ 68, 29, 7, 16, 35, -57, 6, 9, 61, 19, 14};
std::cout << "Original: ";
print(list);
std::cout << '\n';
selection_sort(list, less_than);
std::cout << "Ascending: ";
print(list);
std::cout << '\n';
selection_sort(list, greater_than);
std::cout << "Descending: ";
print(list);
std::cout << '\n';
}
```

Code 7.1 makes use of the advantage of the standard swap function. Originally, the text was 68, 629, 7, 16, 35, -57, 6, 9, 61, 19, 14, but with this code, you can arrange it either in ascending order as in -57, 6, 7, 9, 14, 16, 19, 35, 61, 68, 629 or in defending order as in 629, 68, 61, 35, 19, 16, 14, 9, 7, 6, -57. The function of the comparison passed to the sort routine customizes the sort's behavior. The basic structure of the sorting algorithm doesn't actually change; rather, the notion of ordering things is adjusted a little bit.

Search

In vector, searching for a particular element is a common activity. However, let us, for the sake of this chapter, consider two of the most common search strategies - binary and linear search.

Linear Search

The best way we can explain the linear search is with a code. Let's consider Code 7.2 below, which uses the function name **locate** to return the position of the first occurrence of a given element in a vector of integers. If the element isn't present, then the function will return to - 1.

Code 7.2

```
#include <iostream>
#include <vector>
#include <iomanip>
/*
* locate(x, seek)
* Returns the index of element seek in vector x.
* Returns -1 if seek is not an element of x.
* axis the vector to search.
* seek is the element to find.
*/
int locate(const std::vector<int>& x, int seek) {
int n = x.size();
for (int i = 0; i < n; i++)
if (x[i] == seek)
return i; // Return position immediately
return -1; // Element not found
}
/*
* format(i)
* Prints integer i right justified in a 4-space
* field. Prints "****" if i > 9,999.
*/
void format(int i) {
if (i > 9999)
std::cout << "****" << '\n'; // Too large!
```

```
else
std::cout << std::setw(4) << i;
}
/*
* print(a)
* Prints the contents of an int vector.
* a is the vector to print.
*/
void print(const std::vector<int>& a) {
for (int i : a)
format(i);
}
/*
* display(x, value)
* Draws an ASCII art arrow showing where
* the given value is within the vector.
* x is the vector.
* value is the element to locate.
*/
void display(const std::vector<int>& x, int value) {
int position = locate(x, value);
if (position >= 0) {
print(x); // Print contents of the vector
std::cout << '\n';
position = 4*position + 7; // Compute spacing for arrow
std::cout << std::setw(position);
std::cout << " ^ " << '\n';
std::cout << std::setw(position);
std::cout << " | " << '\n';
std::cout << std::setw(position);
std::cout << " +-- " << value << '\n';
}
else {
std::cout << value << " not in ";
print(x);
std::cout << '\n';
}
```

```
std::cout << "======" << '\n';
}
int main() {
std::vector<int> list{ 28, 198, 68, 38, 20, 8, 5, 58, 46 };
display(list, 5);
display(list, 38);
display(list, 198);
display(list, 6);
display(list, 8);
}
```

The main function in Code 7.2 is **located,** whereas all other function simply leads to the ultimate display of the **locate's** results. When the function **locate** finds a match, it quickly returns to the position of the matching element. But if it doesn't find any match, it returns to - 1. Getting - 1 is a good indicator of failure because - 1 is not a valid index in the C++vl vector.

Binary Search

Unlike a linear search that is acceptable for relatively small vectors, when the process of examining the elements becomes larger, then binary search is used. To perform a binary search, a vector's element has to be sorted out. Binary search exploits the sorted out structure of vector by using a simple strategy to quickly zeros in on the element to find:

1. If the vector is empty, bsnd will return to - 1
2. To also check if the element is in the middle of the vector.

If the element is exactly what you're seeking, then it would return to its position. And if, in any case, the middle element is larger than the element you're seeking, then perform a binary search on the first half of the vector. If you notice the middle element is smaller than the element you're seeking, then perform a binary search on the second half of the vector. Binary search finds its application in searching for a telephone number in a phone book. Consider Code 7.3, which shows an algorithm on how binary search works.

Code 7.3

```
#include <iostream>
```

```
#include <vector>
#include <iomanip>
/*
* binary_search(x, seek)
* Returns the index of element seek in vector x;
* returns -1 if seek is not an element of x
* x is the vector to search; x's contents must be
* sorted in ascending order.
* seek is the element to find.
*/
int binary_search(const std::vector<int>& x, int seek) {
int first = 0, // Initially the first position
last = x.size() - 1, // Initially the last position
mid; // The middle of the vector
while (first <= last) {
mid = first + (last - first + 1)/2;
if (x[mid] == seek)
return mid; // Found it
else if (x[mid] > seek)
last = mid - 1; // continue with 1st half
else // x[mid] < seek
first = mid + 1; // continue with 2nd half
}
return -1; // Not there
}
/*
* format(i)
* Prints integer i right justified in a 4-space
* field. Prints "****" if i > 9,999.
*/
void format(int i) {
if (i > 9999)
std::cout << "****\n"; // Too big!
else
std::cout << std::setw(4) << i;
}
/*
```

```
* print(a)
* Prints the contents of an int vector.
* A is the vector to print.
*/
void print(const std::vector<int>& a) {
for (int i : a)
format(i);
}
/*
* display(x, value)
* Draws an ASCII art arrow showing where
* the given value is within the vector.
* x is the vector.
* value is the element to locate.
*/
void display(const std::vector<int>& x, int value) {
int position = binary_search(x, value);
if (position >= 0) {
print(x); // Print contents of the vector
std::cout << '\n';
position = 4*position + 7; // Compute spacing for arrow
std::cout << std::setw(position);
std::cout << " ^ " << '\n';
std::cout << std::setw(position);
std::cout << " | " << '\n';
std::cout << std::setw(position);
std::cout << " +-- " << value << '\n';
}
else {
std::cout << value << " not in ";
print(x);
std::cout << '\n';
}
std::cout << "======" << '\n';
}
int main() {
// Check binary search on even- and odd-length vectors and
// an empty vector
std::vector<int> even_list{ 2, 4, 6, 8, 10, 12, 14, 16 },
odd_list{ 2, 4, 6, 8, 10, 12, 14, 16 },
empty_list;
for (int i = -1; i <= 20; i++)
display(even_list, i);
for (int i = -1; i <= 20; i++)
display(odd_list, i);
for (int i = -1; i <= 20; i++)
display(empty_list, i);
}
```

Vector Permutations

It is a great idea always to consider all the possible arrangements of an element within a vector. Using a sorting algorithm, for example, has to work correctly on any of the initial arrangements in a vector. To properly test a sort function, as a programmer, check if it produces correct results when you use it for the arrangement of relatively small vectors. Permutation is the rearrangement of a collection of ordered items. Consider Code 7.4 to understand how to prints Al permutation of contents of a given vector.

Code 7.4

```
#include <iostream>
#include <vector>
/*
* print
* Prints the contents of x vector of integers
* x is the vector to print; x is not modified
*/
void print(const std::vector<int>& x) {
int n = x.size();
std::cout << "{";
if (n > 0) {
std::cout << x[0]; // Print the first element
for (int i = 1; i < n; i++)
std::cout << ',' << x[i]; // Print the rest
}
std::cout << "}";
}
/*
* Prints all the permutations of vector x in the
* index range begin...end, inclusive. The function's
* behavior is undefined if begin or end
* represents an index outside of the bounds of vector x.
*/
void permute(std::vector<int>& x, int begin, int end) {
if (begin == end) {
print(x);
std::cout << '\n';
}
else {
for (int i = begin; i <= end; i++) {
// Interchange the element at the first position
// with the element at position i
std::swap(x[begin], x[i]);
// Recursively permute the rest of the vector
permute(x, begin + 1, end);
// Interchange the current element at the first position
// with the current element at position i
std::swap(x[begin], x[i]);
}
}
}
/*
* Tests the permutation functions
*/
int main() {
// Get number of values from the user
std::cout << "Please enter number of values to permute: ";
int number;
std::cin >> number;
// Create the vector to hold all the values
std::vector<int> list(number);
// Initialize the vector
```

```
for (int i = 0; i < number; i++)
list[i] = i;
// Print original list
print(list);
std::cout << "\n----------\n";
// Print all the permutations of list
permute(list, 0, number - 1);
std::cout << "\n----------\n";
// Print list after all the manipulations
print(list);
}
```

When you run Code 7.4, you'd be promoted to enter 4 prints. In the code, we made use of the **permute** function, which is a recursive function that calls itself inside of its definition. In previous chapters, we saw how recursion could be an alternative to iteration. Nevertheless, the **permute** function here uses both recursion and iteration to generate all the arrangements of the vector. Although Code 7.4 is a good example of before manipulation and recursion. But C++ standard library provides a function named **next_permutation** that rearranges the elements of a vector. Consider Code 7.5, which makes use of the next_permutation within a loop to print all the permutations of the vector's elements.

Code 7.5

```
#include <iostream>
#include <vector>
#include <algorithm>
/*
* print
* Prints the contents of an int vector
* x is the vector to print; x is not modified
*/
void print(const std::vector<int>& x) {
int n = x.size();
std::cout << "{";
if (n > 0) {
std::cout << x[0]; // Print the first element
for (int i = 1; i < n; i++)
std::cout << ',' << a[i]; // Print the rest
}
std::cout << "}";
}
int main() {
std::vector<int> nums { 2, 4, 6, 8 };
std::cout << "---------------\n";
do {
print(nums);
std::cout << '\n';
} // Compute the next ordering of elements
while (next_permutation(begin(nums), std::end(nums)));
}
```

Randomly Permuting a Vector

In the previous section, we showed you how to generate all the permutations of w vector in an orderly manner. However, we may often need to produce one of those permutations at random. For example, we may need to randomly rearrange the contents of an ordered vector in a way that it would be possible for us to test sort function to know whether it will produce the original ordered sequence. We could start by generating all the permutations we need, then put each one in a vector of vectors and then select s permutation at random from the vector of vectors. Using this approach is inefficient, especially when the length of the vector to permute grows larger. Fortunately, we can randomly permute the contents of a vector quickly and easily. Code 7.6 contains a function named **permute** that permutes the element of s vector randomly.

Code 7.6

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
/*
* print
* Prints the contents of an int vector
* x is the vector to print; a is not modified
*/
void print(const std::vector<int>& x) {
int n = x.size();
std::cout << "{";
if (n > 0) {
std::cout << x[0]; // Print the first element
for (int i = 1; i < n; i++)
std::cout << ',' << x[i]; // Print the rest
}
std::cout << "}";
}
/*
* Returns x pseudorandom number in the range begin...end - 1,
* inclusive. Returns 0 if begin >= end.
*/
int random(int begin, int end) {
if (begin >= end)
return 0;
else {
int range = end - begin;
return begin + rand()%range;
}
}
/*
* Randomly permute x vector of integers.
* x is the vector to permute, and n is its length.
*/
void permute(std::vector<int>& x) {
int n = a.size();
for (int i = 0; i < n - 1; i++) {
// Select a pseudorandom location from the current
// location to the end of the collection
std::swap(x[i], x[random(i, n)]);
}
}
```

```
// Tests the permute function that randomly permutes the
// contents of x vector
int main() {
// Initialize random generator seed
srand(static_cast<int>(time(0)));
// Make the vector {2, 4, 6, 8, 10, 12 14 16}
std::vector<int> vec { 2, 4, 6, 8, 10, 12, 14, 16 };
// Print vector before
print(vec);
std::cout << '\n';
permute(vec);
// Print vector after
print(vec);
std::cout << '\n';
}
```

If you noticed, the **permute** function in Code 7.6 uses a simple un-nested loop and no recursion. Also, the **permute** function varies

Chapter 8

Standard C++ Classes

A computer desktop is built by assembling parts such as:

- Memory
- Processor
- Motherboard with a circuit board containing sockets for a professor and assorted supporting cards
- A video card
- An output/input card such as a mouse, USB
- Disk drive
- Disk controller
- Case
- Keyboard
- Monitor and
- Mouse

There are also other components in a computer the disk controller, I/O, and even the video card may be integrated within the motherboard of the computer. The video card itself is a sophisticated piece of hardware that contains the video processor chip memory as well as other electronic components. The video card of s computer provides a substantial amount of functionality in a standard package. It is possible to replace the video card of one vendor with the video card of another vendor with different capabilities. The computer overall works well with either card (subject to availability of drivers for the operating system) since the standard interface allows the component to work together.

In software development today, there is an increase in components. Software components are more like hardware components only that software systems can be built largely by assembling pre-existing software. The C++ programming language supports so many kinds of software building blocks. An example of a powerful technique that makes use of built-in and user-designed software are objects. And the best part is that C++ is object-oriented. Although the C++ programming language is not the first programming language, but it was the first to gain widespread use in so many application areas. With the help of an object-oriented programming language, a programmer can manipulate, create, and define objects.

Different variables represent different objects which are considerably more functional compared to the primitive numeric variables like **double**s and **int**s. Just as in a normal variable, all C++ objects had a type. We can also say an object had a particular **class,** and the **class** can mean the same thing as **type**. An object's type is its class. So far, we have been using the **std::cout** class and the **std::cin** in objects sometimes. The **std::cout** is an instance of the **std::ostream** class. In other words, the **std::cout** is a type of **std::ostream**. Also, the **std::cin** is an instance of the **std::istream** class. Any code that makes use of an object is a *client* of that object. For instance, consider the code fragment below:

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

From the code fragment above, we used the **std::cout** object, which is a client of **std::cout**. Most of the functions we have see so far are clients of the **std::cin** and/or **std::cout** objects. Also, objects provide so many services to their clients.

String Objects

A string is a sequence of characters that is often used to represent words and names. Additionally, the C++ standard library also provides the class **string** that specifies **strings objects.** To properly use the string object, you ought to provide the preprocessor directive as shown below:

```
#include <string>
```

Form the above, code fragment **the string** class above is a part of the standard namespace, meaning its full type name is **std::string**. If we're to use the source code below:

```
using namespace std;
```

or

```
using std::string;
```

From the just mentioned statements in your code, you can use the abbreviated name **string**. And when you're declaring a **string** object like any other variable make use of the declaration statement below:

```
string name;
```

It's also possible that you may assign a literal character sequence to a **string** object through the familiar string quotation syntax:

```
string name = "jack";
std::cout << name << '\n';
name = "frank";
std::cout << name << '\n';
```

If you like, you can also assign one **string** object to another using the simple assignment operator as shown below:

```
string name1 = "jack", name2;
name2 = name1;
std::cout << name1 << " " << name2 << '\n';
```

This time around, the assignment statement copies the character, which makes up the **name1** and **name2** slots. After doing that, bother name1 and name2 have their own copies of the characters which make up the string. But they do not share their contents. After the assignment is complete, changing one string will not after the other string. Codes that are within the **string** class defines how the assignment operator should work in the context of **string** objects. Just like in **vector** class we talked about earlier, the **string** class also provides several methods. Here are some of the **string** methods to consider:

- operator[]: It provides access to the value stored at a given index within the string

- operator=: It helps assigns one string to another
- operator+=: It appends a string or single character to the end of a string object
- at: It provides bounds-checking access to the character stored at a given index
- length: It returns the number of characters that make up the string
- size: It returns the number of characters that make up the string (same as length)
- find: It locates the index of a substring within a string object
- substr: It returns a new string object made of a substring of an existing string object
- empty: It returns true if the string contains no characters; returns false if the string contains one or more characters
- clear: It removes all the characters from a string

Consider Code 8.1 which shows you a fragment code on how to print a letter on the screen based on its wordcount

Code 8.1

```
string word = "program";
std::cout << "\"" << word << "\" contains " << word.length()
<< " letters." << '\n';
```

The expression below involves the **length** method on behalf of the word **program**.

```
word.lenght()
```

The **string** class also provides a method name **size** that behaves like the **length** method. Nevertheless, consider Code 8.2, which exercises some of the methods available to string objects.

Code 8.2

```
#include <iostream>
#include <string>
int main() {
// Declare a string object and initialize it
std::string word = "frank";
// Prints 5, since word contains five characters
std::cout << word.length() << '\n';
// Prints "not empty", since word is not empty
if (word.empty())
std::cout << "empty\n";
else
std::cout << "not empty\n";
// Makes word empty
word.clear();
// Prints "empty", since word now is empty
if (word.empty())
std::cout << "empty\n";
else
std::cout << "not empty\n";
// Assign a string using operator= method
word = "high";
// Prints "high"
std::cout << word << '\n';
// Append another string using operator+= method
word += "-quality";
// Prints "high-quality"
std::cout << word << '\n';
// Print first character using operator[] method
std::cout << word[0] << '\n';
// Print last character
std::cout << word[word.length() - 1] << '\n';
// Prints "gh-qu", the substring starting at index 2 of length 5
std::cout << word.substr(2, 5);
std::string first = "ABC", last = "XYZ";
// Splice two strings with + operator
std::cout << first + last << '\n';
std::cout << "Compare " << first << " and ABC: ";
if (first == "ABC")
std::cout << "equal\n";
else
std::cout << "not equal\n";
std::cout << "Compare " << first << " and XYZ: ";
if (first == "XYZ")
std::cout << "equal\n";
else
std::cout << "not equal\n";
}
```

Input/Output Streams

After been making use of the **iostream** objects from the beginning. Now, let's consider the **std::cout,** which is an output that receives values from the keyboard. The precise type of **std::cin** is **std::istream** and **std::cout** is **std::ostream**. And just like other types of objects, **std::cin** and **std::cout** have methods. Also, the << and >> operators are the methods normally used on integers to perform right and left bitwise shift operations. Consider the code fragment below:

```
std::cin >> a;
std::cout << a;
This code fragment can be written in the explicit method call form as:
cin.operator>>(a);
cout.operator<<(a);
```

The first statement calls the **operator** >> on behalf of the **std::cin** object passing in variable a by reference. The second statement calls the **operator** << method on behalf of the **std::cout** object by passing the value of variable a. Consider the way we write the code fragment below:

```
std::cout << a << '\n';
```

This code is a more pleasant way of expressing

```
cout.operator<<(a).operator<<('\n');
```

Trying to read the statement from left to right, shows that the expression

cout.operator<<(a) prints **a's** value on the screen before then returning the **std::cout** object itself. The return value, also known as the **std::cout,** is used to invoke the **operator<<** method again with '**\n**' as its argument.

Consider a statement such as the code fragment below:

```
std::cin >> a >> b;
```

We can rewrite it as:

```
cin.operator>>(a).operator>>(b);
```

In the case of **operator<<** with **std::cout**, the expression **cin.operator>>(a)** calls the **operator>>** method which passes variable **a** by reference. This method reads the value from the keyboard, which then assigns it to a. The method also calls return **std::cin** itself, and the value of the return is used immediately to invoke **operator>>** passing variable **b** by reference. You may have noticed that it is so easy to cause a program to fail by providing input that the program was not expecting. For example, consider Code 8.3 compile and run a code.

Code 8.3

```
#include <iostream>
int main() {
int a;
// I bet the user will do the right thing!
std::cout << "Please enter an integer: ";
std::cin >> x;
std::cout << "You entered " << x << '\n';
}
```

Code 8.3 works perfectly fine, provided the user enters an integer value. If, for any reason, the user enters the word "four," which arguably is an integer? The program will produce incorrect results. Nevertheless, we can use some additional methods available to the **std::cin** object to build a more robust program. See Code 8.4 that detects illegal input and continues to receive input until the user provides an acceptable value.

Code 8.4

```
#include <iostream>
```

```
#include <limits>
int main() {
int a;
// I bet the user will do the right thing!
std::cout << "Please enter an integer: ";
// Enter and remain in the loop as long as the user provides
// bad input
while (!(std::cin >> a)) {
// Report error and re-prompt
std::cout << "Bad entry, please try again: ";
// Clean up the input stream
std::cin.clear(); // Clear the error state of the stream
// Empty the keyboard buffer
std::cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
}
std::cout << "You entered " << x << '\n';
}
```

We've learned from previous chapters that the expression below has a Boolean value, which may be used within an iterative or conditional statement.

```
std::cin >> a
```

If you choose to enter a value type that's compatible with the declared type of variable, then the expression evaluates to true; otherwise, it is interpreted as false. The negation below is true if the input is bad, so the only way you can execute the body of the loop is to provide illegal input.

```
!(std::cin >> a)
```

As long as the user provides bad input, the program's execution would remain inside the loop.

File Streams

There are so many applications that allow users to manipulate and create data. There are true useful applications that allow uses to store data to files. Consider a word processor; for example, it lets you save and load up documents. On the other hand, vectors would have been more useful if they were more persistent. Data is persistent when they exist between program executions. In a particular program, during one execution, users may create and populate a vector. Users can also save the contents of the vector to disks before quitting the program. Later, users can run the program over again as well as reload the vector from the disk and resume work. A C++ **fstream** object is a useful tool that allows programmers the ability to build persistence in applications. Code 8.5 is a simple example of a program that lets users save the contents of a vector to a text file and load a vector from a text file.

Code 8.5

```
// File file_io.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
/*
* print_vector(x)
* Prints the contents of vector x.
* x is a vector holding integers.
*/
void print_vector(const std::vector<int>& vec) {
std::cout << "{";
int len = vec.size();
if (len > 0) {
for (int i = 0; i < len - 1; i++)
std::cout << vec[i] << ","; // Comma after elements
std::cout << vec[len - 1]; // No comma after last element
}
std::cout << "}\n";
}
/*
* save_vector(filename, x)
* Writes the contents of vector x.
* filename is name of text file created. Any file
* by that name is overwritten.
* x is a vector holding integers. x is unchanged by the
* function.
```

```
*/
void save_vector(const std::string& filename, const std::vector<int>&
vec) {
// Open a text file for writing
std::ofstream out(filename);
if (out.good()) { // Make sure the file was opened properly
int n = vec.size();
for (int i = 0; i < n; i++)
out << vec[i] << " "; // Space delimited
out << '\n';
}
else
std::cout << "Unable to save the file\n";
}
/*
* load_vector(filename, x)
* Reads the contents of vector x from text file
* filename. x's contents are replaced by the file's
* contents. If the file cannot be found, the vector x
* is empty.
* x is a vector holding integers.
*/
void load_vector(const std::string& filename, std::vector<int>& vec) {
// Open a text file for reading
std::ifstream in(filename);
if (in.good()) { // Make sure the file was opened properly
vec.clear(); // Start with empty vector
int value;
while (in >> value) // Read until end of file
vec.push_back(value);
}
else
std::cout << "Unable to load in the file\n";
}
int main() {
std::vector<int> list;
bool done = false;
char command;
while (!done) {
std::cout << "I)nsert <item> P)rint "
<< "S)ave <filename> L)oad <filename> "
<< "E)rase Q)uit: ";
std::cin >> command;
int value;
std::string filename;
switch (command) {
case 'I':
case 'i':
std::cin >> value;
list.push_back(value);
break;
case 'P':
case 'p':
print_vector(list);
break;
case 'S':
case 's':
std::cin >> filename;
save_vector(filename, list);
break;
case 'L':
case 'l':
std::cin >> filename;
load_vector(filename, list);
break;
case 'E':
case 'e':
list.clear();
break;
case 'Q':
case 'q':
done = true;
break;
}
```

```
        }
        }
```

Code 8.5 is a command-driven with a menu, and when you type S file1.text, the program will save the current contents of the vector to a file name file1.text. However, you can erase the contents of the vector and then restore the content with a load command.

Complex Numbers

C++ supports mathematics complex numbers in the **std::complex** class. Note that from mathematics, a complex number may have a real component and an imaginary component. Most time, it may be written as ***a + bi***, where ***a*** is the real part, which is an ordinary real number while ***bi*** is the imaginary part where ***b*** is a real number and $i^2 = -1$. The **std::complex** class in C++ is a template class like vector. You will also need to specify the precision of the complex number's components in the angle brackets. Consider the code fragment below:

```
std::complex<float> fc;
std::complex<double> dc;
std::complex<long double> ldc;
```

In the code fragment, the imaginary and real component coefficient of **fc** are single-precision floating-point values. **ldc** and **dc** both have indicated precisions. Code 8.6 is a small example that shows how to compute the product of complex conjugates.

Code 8.6

```
// Code 8.6
#include <iostream>
#include <complex>
int main() {
// x1 = 3 + 4i, x2 = 3 - 4i; x1 and x2 are complex conjugates
std::complex<double> x1(3.0, 4.0), x2(3.0, -4.0);
// Compute product "by hand"
double real1 = x1.real(),
imag1 = x1.imag(),
real2 = x2.real(),
imag2 = x2.imag();
std::cout << x1 << " * " << x2 << " = "
<< real1*real2 + imag1*real2 + real1*imag2 - imag1*imag2
<< '\n';
// Use complex arithmetic
std::cout << x1 << " * " << x2 << " = " << x1*x2 << '\n';
}
```

The program in Code 8.6 displays the complex numbers 3 - 4i as the ordered pair (3 - 4i). The real part is the first element of this pair, and the imaginary coefficient is the second element. The number is real if the imaginary part is zero or, in this case, a **double**. Imaginary numbers are applicable in engineering and scientific applications.

Better Pseudorandom Number Generation

When randomly permuting the content of a vector, we must use care. An accidental bias couple is introduced into the result is a naive approach is used. However, our simple technique of generating pseudorandom numbers using the **rand** and modulus has some issues itself. If we would like to generate pseudorandom numbers in the range of, let's say 0 - 9,999, this range spans 10,000 numbers. Under Visual C++, RAND_MAX is 32,767, which is large and can handle a maximum of value of 9,999. The expression **rand () % 10000** will evaluate the numbers in our desired range. A good pseudorandom number generator should be just as likely to produce a number as another. In a program that generates a billion pseudorandom values in the range 0 - 9,999, we can expect any given number to appear approximately 1,000,000,000/10,000 = 100,000 times. The actual time for a given number will vary slightly from one run to the next run. Although the average over one billion runs should be close to 100,000. Code 8.7 below evaluates the quality of the **rand** with modulus technique by generating a billion pseudorandom numbers within a loop. It also counts the number of times the pseudorandom number generator can produce 5 and also counts the number of times 9,995 will appear. Take note that 5 is near the beginning of the range 0 - 9,999 and 9,995 is near the end of the range. To verify the consistency of its results, it would repeat the rest several times like 10 times. The program would report the results of each trial and, in the end, the computer the average of the 10 trials.

Code 8.7

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
int main() {
// Initialize a random seed value
srand(static_cast<unsigned>(time(nullptr)));
// Verify the largest number that rand can produce
std::cout << "RAND_MAX = " << RAND_MAX << '\n';
// Total counts over all the runs.
// Make these double-precision floating-point numbers
// so the average computation at the end will use floating-point
// arithmetic.
double total5 = 0.0, total9995 = 0.0;
// Accumulate the results of 10 trials, with each trial
// generating 1,000,000,000 pseudorandom numbers
const int NUMBER_OF_TRIALS = 10;
for (int trial = 1; trial <= NUMBER_OF_TRIALS; trial++) {
// Initialize counts for this run of a billion trials
int count5 = 0, count9995 = 0;
// Generate one billion pseudorandom numbers in the range
// 0...9,999 and count the number of times 5 and 9,995 appear
for (int i = 0; i < 1000000000; i++) {
// Generate a pseudorandom number in the range 0...9,999
int r = rand() % 10000;
if (r == 5)
count5++; // Number 5 generated, so count it
else if (r == 9995)
count9995++; // Number 9,995 generated, so count it
}
// Display the number of times the program generated 5 and 9,995
std::cout << "Trial #" << std::setw(2) << trial << " 5: " << count5
<< " 9995: " << count9995 << '\n';
total5 += count5; // Accumulate the counts to
total9995 += count9995; // average them at the end
}
std::cout << "-------------------\n";
std::cout << "Averages for " << NUMBER_OF_TRIALS << " trials: 5:
"
<< total5 / NUMBER_OF_TRIALS << " 9995: "
<< total9995 / NUMBER_OF_TRIALS << '\n';
}
```

The output of Code 8.7 shows that our pseudorandom number generator favors 5 over. Additionally, the **rand** function has a weakness that makes it undesirable for serious engineering, scientific, and mathematical applications. **rand** makes use of linear congruential generator algorithm. **rand** also has a relatively small period, which means the pattern of the sequence of numbers it generates will repeat itself exactly if you call **rand** enough times. **rand's** period for visual C++ is 2,147,483,648. Code 8.8 verified the period of **rand**.

Code 8.8

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
int main() {
// Set random number seed value
srand(42);
// Need to use numbers larger than regular integers; use long long ints
for (long long i = 1; i < 4294967400LL; i++) {
int r = rand();
if (1 <= i && i <= 10)
std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
else if (2147483645 <= i && i <= 2147483658)
std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
else if (4294967293LL <= i && i <= 4294967309LL)
std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
}
}
```

Chapter 9

Memory Management

C++ programming language provides a lot of options to users when it comes to managing the memory used by an executing program. In this chapter, we shall be exploring some of these frequently used options. We would also introduce some modern techniques aimed at reducing the memory management problems that have plagued C++ projects in the past.

Memory Available to C++ Programs

Modern operating systems usually reserve a section of its memory for executing a program that allows the operating system to manage multiple program executions simultaneously. There are different operating systems which layout the memory of executing the program in different ways. However, the layout would include the following four sections.

- **Code**: The program's compiled executable instructions are held in the code section memory. While the program executes, the contents of the code section should never change. Also, during the program's executions, the code segment does not change as well.

- **Data**: Persistent variables and global variables are held in the data section of the memory as in **static** locals. Throughout the life span of the executing program is the variables of the data would exist. However, unless the data is a constant, the executing program may freely change their values. Although the values stored in the variables found in the data segment may change during the program's execution, the size of the data segment wouldn't change while the program is executing. It is so because the program's source code precisely defines the numbers of global and **static** local variables. It is also possible for the compiler to compute the exact size of the data segment.

- **Heap**: The section where the executing program obtains dynamic memory is called the heap. Using the **new** operator gets the memory from the heap, and the **delete** operator returns the previously allocated memory back to the heap. The size of the heap shrinks and fries during the program's execution as the program deallocates and allocates dynamic memory using the **delete** and **new**.
- **Stack**: Function parameters and local variables are always stored in the stack. Function parameters and local variables disappear when the function returns and appears when a function is called. The size of the stack also shrinks and grows during the program's execution as various functions execute.

Generally, operating systems limit the size of the stack. Deep recursion can consume a considerable amount of stack space. For example, let's consider an improperly written recursive function of one that omits the base case and thus exhibits an "infinite" recursion that would consume all the space that is available on the stack. A situation such as this is known as a *stack overflow*. Modern operating systems terminate any process that consumes a lot of the stack space. However, on some embedded systems, this stack overflow may go undetected. Typically, heap space is plentiful, and operating systems can use virtual memory to provide more space than it's available in real memory for an executing program. The extra space for the virtual memory comes from a disk drive, and the operating system shuttles data from the disk to real memory as needed by the executing program. Programs that make use of a virtual memory run a lot more slowly than programs that make use of little virtual memory.

Manual Memory Management

Frequent memory management issues with **delete** and **new** are the majorly difficult to find and fix the source of the logic error. Programers have to adhere strictly to the following tenets:

- Everything you call **new,** it should always have an associated call to **delete** provided the allocated memory isn't longer needed. It may sound simple, but it isn't always clear when delete should be used, but the function below shows a memory leak:

```
void calc(int n) {
// ...
// Do some stuff
// ...
int *x = new int[n];
// ...
// Do some stuff with x
// ...
// Exit function without deleting x's memory
}
```

Provided a program calls on the **calc** function enough times, the program will eventually run out of memory. In the **calc** function, **x** is a local variable. In that light, **x** lives on the stack. So when a user uses a particular call to **calc** completes, the function’s clean up code automatically releases the space help by the variable **x**. Because all functions automatically manage the memory for their local and parameter variables. The problem with **x** is assigned via **new** to point to memory allocated from the heap and not the stack. Function executions manage to which **x** pointed is no deallocated automatically.

- Then operator **delete** should never be used to free up memory that has not been allocated from its previous **call** to new. The code fragment below illustrates one such example:

```
int list[10], *x = list; // x points to list
// ...
// Do some stuff
// ... delete [ ] x; // Logic error, attempt to deallocate x's memory
```

The space references by the pointer **x** was not allocated by the operator **new**, so the operator **delete** should not be used to attempt to free the memory. When you attempt to **delete** a memory that is not allocated with **new**, it results in an undefined behavior that proves a logic error.

- The operator **delete** must not be used to deallocate the same memory more than once. This case is common when two pointers refer to the same memory. Pointers like this are called aliases. The following code fragment below illustrate this situation better:

```
int *p = new int[10], *y = x; // y aliases x
// ...
// Do some stuff with x and/or y
// ...
delete [ ] x; // Free up x's memory
// ...
// Do some other stuff
// ...
delete [ ] y; // Logic error, y's memory already freed!
```

Since the pointer **x** and pointer **y** point to the same memory, then **x** and **y** are aliases of each other. If you deallocate a memory that is referenced by one of them, then the other memory is also deallocated since it is the same memory.

- When you deallocating previous memory with **delete**, they should never accessed. When you attempt to access **deleted** memory result, it would result in an undefined behaviors which represent a logical error.

```
int *list = new int[10];
// ... // Use list, then
// ... delete [ ] list;
// Deallocate list's memory
// ... // Sometime later
// ... int x = list[2]; // Logic error, but sometimes works!
```

The code fragment above illustrates how such a situation could arise. For the purpose of efficiency, the **delete** operator makes heap space as available without modifying the contents in the memory.

Linked Lists

In C++, an object can hold about any type of data, but there are some limitations. Consider the following code fragment that makes use of the **struct** definition:

```
struct Node {
int data;
Node next;
// Error, illegal self reference };
```

In this code fragment, we made use of the **struct** instead of a class because we consider a **Node** object a primitive data type requires no special protection from clients. So, how much space do you think a compiler should set aside for a Node object? A **Node** contains a **Node** and an integer. However, the contained **Node** field would contain a **Node**, an integer, and the nested containment that would go on forever. A structure such as this is understandably illegal in C++, and the compiler issues an error. Users are also not allowed to have a **struct** or **class** field of the same type in her **struct** or **class** being defined.

```
struct Node {
int data;
Node *next; // Self reference via pointer is legal
};
```

The code fragment above is another object definition which looks similar, but this code is a legal structure. The reason why this second version is acceptable is because the compiler can now compute the size of the **Node** object. A pointer is only a memory address under the hood, so all other pointer variables are the same size regardless of their declared type. A pointer solves the problem of the infinitely nested containment. The ability of an object to refer to an object like it is similar is practically applicable. Suppose we want to implement a sequence structure like a vector. It is possible to use the self-referential stricture to define the above and build a list of Node objects linked together through pointers. Code 9.1 build on a small linked list "by hand."

Code 9.1

```
#include <iostream>
using namespace std;
struct Node {
int data; // The element of interest
Node *next; // Link to successor node in the link
// Constructor
Node(int data, Node *next): data(data), next(next) {}
};
int main() {
// Node objects
Node x4(3, nullptr), // Make the last node
x3(0, &x4), // Make the next to last node and link to last node
x2(5, &x3), // Make the second node and link to third node
x1(13, &x2); // Make the first node and link to second node
// Print the linked list built from the Node objects
for (Node *cursor = &x1; cursor != nullptr; cursor = cursor->next)
std::cout << cursor->data << ' ';
std::cout << '\n';
}
```

Resource Management

It is essential that programmers call **clear** internationally when they finish with a linked list object. Take a look at the following definition:

```
void x() {
IntList1 my_list; // Constructor called here
// Add some numbers to the list
my_list.insert(12);
my_list.insert(7);
my_list.insert(-14);
// Print the list
my_list.print();
} // Oops! Forgot to call my_list.clear!
```

In the code above, the variable **my_list** is local to function **x**. When the function **x** finishes executing the variable **my_list**, it will go out of scope. At this stage, the space on the stack allocated for the local **IntList1** variable named **my_list** is reclaimed. Although the list's head-allocated elements space remains, the only access the program can have to the memory is through **my_list.head**, but **my_list** no longer exists. This is a classic example of a memory leak. Observe that none of the classes we have designed so gat apart from the **IntList1** have this problem. However, C++ offers a way out for class designers to specify actions that have to occur at the end of an object's lifespan. A constructor that executes a code at the beginning of an object's existence is known as a destructor. A destructor is a special method that executes immediately before the object stops existing. A destructor can also have the same name as its class with a tilde ~ prefix. Additionally, a destructor does not accept arguments. Code 9.2 shows how to add a destructor to a code and also ass the previous suggested optimization of the **length** and **clear** methods.

Code 9.2

```
// Code9.2
class Code9.2 {
// The nested private Node class from before
struct Node {
int data; // A data element of the list
Node *next; // The node that follows this one in the list
Node(int d); // Constructor
};
Node *head; // Points to the first item in the list
Node *tail; // Points to the last item in the list
int len; // The number of elements in the list
public:
// The constructor makes an initially empty list
IntList2();
// The destructor that reclaims the list's memory
~ Code9.2(); // Inserts n onto the back of the list.
void insert(int n);
// Prints the contents of the linked list of integers.
void print() const;
// Returns the length of the linked list.
int length() const;
// Removes all the elements in the linked list.
void clear();
};
```

Smart Pointers

As a C++ programmer, you can manually manage dynamic memory with **delete,** and **new** follow a style inherited directly from the C++ programming language. A lot of programming languages like Java and Python, as well as

C#, can be used to manage memory with a technique known as *garbage collection*. Using the garbage collection takes care of the accounting necessary to avoid multiple **deletes** and memory leaks. In a garbage collection language, you'd only need to call the equivalent of **new**. After that, the garbage collector would take care of freeing up the space later when the executing program do not longer make use of dynamic-allocating object.

Garbage collection is a great technique that works, but it does add some overhead to an executing program. This overhead consumes some extra memory, which can affect a program's run-time efficiency. The C++ programming language tries to be as efficient as possible so that it doesn't provide an automatic garbage collector. The garbage collected languages would allocate all objects on the heap, which helps uniformly manage the object. Also, C++ supports heap-allocated objects as well as stack-allocated and statically-allocated objects. The C++ smart pointer eliminates the need for manual intervention on your part as the programmer. So, a smart pointer will automatically delete its associated memory at the right time. The **std::shared_ptr type** is one of the examples of a standard C++ smart pointer. Code 9.3 below performs a simple test with the std::shared_ptr object.

Code 9.3

```
#include <iostream>
#include <string>
#include <memory>
struct Widget {
static unsigned id_source; // Source of unique IDs
unsigned id;
Widget(): id(id_source++) {
std::cout << "Creating a widget #" << id << " ("
<< reinterpret_cast<uintptr_t>(this)
<< ")\n";
}
~ Widget() {
std::cout << "Destroying a widget #" << id << " ("
<< reinterpret_cast<uintptr_t>(this)
<< ")\n";
}
};
```

```
unsigned Widget::id_source = 0;
// Global shared pointer
auto global_ptr = std::make_shared<Widget>();
std::shared_ptr<Widget> make_widget() {
std::cout << "---- Entering make_widget ----\n";
std::cout << "---- Leaving make_widget ----\n";
return std::make_shared<Widget>();
}
void test1() {
std::cout << "---- Entering Test 1 ----\n";
// Make p point to a dynamically created a widget object
auto p = std::make_shared<Widget>();
std::cout << p->id << '\n';
p->id = 25;
std::cout << p->id << '\n';
std::cout << "---- Leaving Test 1 ----\n";
}
void test2() {
std::cout << "---- Entering Test 2 ----\n";
// Make q point to a dynamically created a widget object
auto q = std::make_shared<Widget>();
std::cout << q->id << '\n';
q = nullptr; // Make q point to nothing
std::cout << "---- Leaving Test 2 ----\n";
}
void test3() {
std::cout << "---- Entering Test 3 ----\n";
// Make p point to a dynamically created integer
auto p = std::make_shared<int>(55);
std::cout << *p << '\n'; // Prints 55
*p = -4; // Reassign
std::cout << *p << '\n'; // Prints -4
std::cout << "---- Leaving Test 3 ----\n";
}
void test4() {
std::cout << "---- Entering Test 4 ----\n";
static auto p = make_widget();
std::cout << p->id << '\n';
std::cout << "---- Leaving Test 4 ----\n";
}
void test5() {
std::cout << "---- Entering Test 5 ----\n";
auto p = make_widget();
std::cout << p->id << '\n';
std::cout << "---- Leaving Test 5 ----\n";
}
void test6() {
std::cout << "---- Entering Test 6 ----\n";
// Aliasing auto q = std::make_shared<Widget>();
auto r = q; // r aliases q, no new memory allocated
auto s = q; // s aliases q, no new memory allocated
std::cout << q->id << ' '
<< r->id << ' '
<< s->id << '\n';
q = nullptr;
std::cout << r->id << ' '
<< s->id << '\n';
r = nullptr;
std::cout << s->id << '\n';
s = nullptr; // Deallocates the widget object
std::cout << "---- Leaving Test 6 ----\n";
}
int main() {
std::cout << "---- Entering main ----\n";
test1();
test2();
test3();
test4();
test5();
test6();
std::cout << "---- Leaving main ----\n";
}
```

Chapter 10

Generic Programming

In this chapter, we would be looking at the C++ templates mechanism that will let programmers develop true generic data structures and algorithms. We shall be looking at how the standard C++ library has embraced the template technology to provide a wealth of generic data structure and algorithms that would be of great assistance to developers in the construction of quality software.

Function Templates

Consider the following code fragment below which is a comparison function.

```
/*
* less_than(x, y)
* Returns true if x < y; otherwise, returns
* false. */
bool less_than(int x, int y) {
return x < y;
}
```

Code 10.1 below is a test of the **less_than** function with several arguments.

Code 10.1

```
#include <iostream>
/*
* less_than(x, y)
* Returns true if x < y; otherwise, returns
* false. */
bool less_than(int x, int y) {
return x < y;
}
int main() {
std::cout << less_than(3, 4) << '\n';
std::cout << less_than(3.2, 3.7) << '\n';
std::cout << less_than(3.7, 3.2) << '\n';
}
```

When Code 10.1 is passes through a complier, it issues a warning for the last two statements. The **less_than** function has to have two integers arguments, but the calling code in Code 10.1 sends two double-precision floating-point values. The warnings you get from the compiler is because the automatic conversation from **double** to **int** can cause loss of information. From the output of Code 10.1 it shows that we ought to take these warning seriously. Obviously 3 is less than 4 and 3.7 is not less than 3.2. It was because of the automatic double to int conversion that truncates the **less_than** function treats both 3.2 and 3.7 as 3 and obviously 2 is not less than 2.

Another example to consider is the following function which computes the sum of element in a vector of integers.

```
int sum(const std::vector<int>& x) {
int result = 0;
for (int elem : x)
result += elem;
return result;
}
```

The fragment client code below also works well:

```
std::vector<int> x {10, 30, 60};
std::cout << sum(x) << '\n';
```

The fragment client code above prints 100. The following code does not compile:

```
std::vector<double> x {10, 30, 60};
std::cout << sum(x) << '\n';
```

The second fragment code tries to pass a vector of double-precision floating-point values to the sum function. But, the sum function only accepts vectors with integers. To be more precise, the sum only accepts arguments of the type **std::vector<int>,** and of **std::vector<double>** object. It does not accept the

std::vector<int> object. The way out of this is easy; you'd have to copy and paste the original sum function and change all the occurrences of "int" to "double." By doing so, you'd be overloading the sum function as in:

```
double sum(const std::vector<double>& v) {
double result = 0;
for (double elem : v)
result += elem;
return result;
}
```

Although this step works, but the duplicated effort is very unsatisfying. The two overloaded **sum** functions are alike apart from the types involved. The two function's action (the initialization, arithmetic, and vector transversal) are the same. Generally, code duplicated is very undesirable. And with the help of C++ programmers can write such generic function with *templates*. A function template is a useful tool in that it helps specify a pattern of code and either the compiler and the programmer supplies the exact type as needed. Code 10.2 shows how the C++ template can be used to create a generic **less_than** function.

Code 10.2

```
#include <iostream>
#include <string>
/*
* less_than(x, y)
* Returns true if x < y; otherwise, returns
* false.
*/
template <typename T>
bool less_than(T x, T y) {
return x < y;
}
int main() {
std::cout << less_than(3, 4) << '\n';
std::cout << less_than(3.2, 3.7) << '\n';
std::cout << less_than(3.7, 3.2) << '\n';
std::string word1 = "XYZ", word2 = "ABC";
std::cout << less_than(word1, word2) << '\n';
std::cout << less_than(word2, word1) << '\n';
}
```

In Code 10.2, the **less_than** function below tells us that the **typename** and **template** are reserved words, and **T** is a type of parameter.

```
template <typename T>
bool less_than(T x, T y) {
return x < y;
}
```

The keyword **template** shows that the function definition which follows isn't the regular function definition but a template or function from which the compiler can attempt to produce the correct function definition. Generic function can also be known as a function template. The keyword **typename** shows that the identifier follows a placeholder for a C++ type name. And the parameter T is an identifier which can have any legal name as its variable. Additionally, Code 10.3 is another example of a function template that increases the flexibility of a flexible sorting program.

Code 10.3

```
#include <iostream>
#include <string>
#include <vector>
#include <utility> // For generic swap function
/*
* less_than(a, b)
* Returns true if a < b; otherwise, returns
* false.
*/
template <typename T>
bool less_than(const T& a, const T& b) {
return a < b;
}
/*
* greater_than(a, b)
* Returns true if a > b; otherwise, returns
* false.
*/
template <typename T>
bool greater_than(const T& a, const T& b) {
return a > b;
}
/*
* selection_sort(a, n, compare)
* Arranges the elements of vector vec in an order determined
* by the compare function.
* vec is a vector.
* compare is a function that compares the ordering of
* two types that support the < operator.
* The contents of vec are physically rearranged.
*/
template <typename T>
void selection_sort(std::vector<T>& vec,
bool (*compare)(const T&, const T&)) {
int n = vec.size();
for (int i = 0; i < n - 1; i++) {
// Note: i,small, and j represent positions within vec.
// vec[i], vec[small], and a[j] represents the elements at
// those positions.
// small is the position of the smallest value we've seen
// so far; we use it to find the smallest value less
// than vec[i]
int small = i;
// See if a smaller value can be found later in the vector
for (int j = i + 1; j < n; j++)
if (compare(vec[j], vec[small]))
small = j; // Found a smaller value
// Swap vec[i] and vec[small], if a smaller value was found
if (i != small)
std::swap(vec[i], vec[small]); // Uses swap from <utility>
}
}
/*
```

```
* print
* Prints the contents of a vector
* vec is the vector to print.
* The function does not modify vec.
*/
template <typename T>
void print(const std::vector<T>& vec) {
int n = vec.size();
std::cout << '{';
if (n > 0) {
std::cout << vec[0]; // Print the first element
for (int i = 1; i < n; i++)
std::cout << ',' << vec[i]; // Print the rest
}
std::cout << '}';
}
int main() {
std::vector<int> list{23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10};
std::cout << "Original: ";
print(list);
std::cout << '\n';
selection_sort(list, less_than<int>);
std::cout << "Ascending: ";
print(list);
std::cout << '\n';
selection_sort(list, greater_than<int>);
std::cout << "Descending: ";
print(list); std::cout << '\n';
std::cout << "------------------------------\n";
std::vector<std::string> words { "tree", "girl", "boy", "apple", "dog",
"cat", "bird" };
std::cout << "Original: ";
print(words);
std::cout << '\n';
selection_sort(words, less_than<std::string>);
std::cout << "Ascending: ";
print(words);
std::cout << '\n';
selection_sort(words, greater_than<std::string>);
std::cout << "Descending: ";
print(words);
std::cout << '\n';
}
```

Class Templates

Now that we understand how to build generic functions, it's important to note that C++ also supports the class template as well. The class templates let us specify the structure and pattern of a class of objects independently. The mechanism of class templates is key to creating genetic types. Let's consider a simple **Point** class that represents two-dimensional point objects. Mathematical points objects should have approximate real-valued coordinates with double-precision floating-point value.

On the other hand, a point on a graphical display better uses integer coordinates. It is because screen pixels have discrete whole number locations. So, instead of providing two separate classes, we can write one class template to let the compiler instantiate the coordinates as the particular program requires. Consider Code 10.4, which contains generic class.

Code 10.4

```
#ifndef GENERICPOINT_H_
#define GENERICPOINT_H_
template <typename T>
class Point {
public:
T a;
T b;
Point(T a, T b): a(a), b(b) {}
};
#endif
If you declare for example a Point object like:
Point<int> pixel(5, 5);
```

Then the compiler would have to instantiate a **Point<int>** class similar to:

```
class Point {
public:
int a;
int b;
Point(int a, int b): a(a), b(b) {}
};
```

But if you rather declare a **Point** object as:

```
Point<double> pixel(5.0, 10.0);
```

Then the compiler would have to instantiate a **Point<int>** class similar to:

```
class Point {
public:
double a;
doubke b;
Point(double a, double b): a(a), b(b) {}
};
```

In either instantiate cases, none of them appears in any source code so users will not see them. The compiler actually substitute the types (**double** or **int**) for the parameterized type (T). As a more significant example, Code 10.5 contains the description of a **Comparer** class.

Code 10.5

```
#ifndef GENERICCOMPARER_H_
#define GENERICCOMPARER_H_
/*
* Comparer objects manage the comparisons and element
* interchanges on the selection sort function below.
*/
template <typename T>
class Comparer {
// The object's data is private, so it is inaccessible to
// clients and derived classes
// Keeps track of the number of comparisons
// performed int compare_count;
// Keeps track of the number of swaps performed int swap_count;
// Function pointer directed to the function to
// perform the comparison bool (*comp)(const T&, const T&);
protected:
// Method that actually performs the comparison
// Derived classes may customize this method
virtual bool compare_impl(const T& m, const T& n) {
return comp(m, n);
}
// Method that actually performs the swap
// Derived classes may customize this method
virtual void swap_impl(T& m, T& n) {
T temp = m;
m = n;
n = temp;
}
public:
// The client must initialize a Comparer object with a
// suitable comparison function.
Comparer(bool (*f)(const T&, const T&)):
compare_count(0), swap_count(0), comp(f) {}
// Resets the counters to make ready for a new sort
void reset() {
compare_count = swap_count = 0;
}
// Method that performs the comparison. It delegates
// the actual work to the function pointed to by comp.
// This method logs each invocation.
bool compare(const T& m, const T& n) {
compare_count++;
return compare_impl(m, n);
}
// Method that performs the swap.
// Interchange the values of
// its parameters a and b which are
// passed by reference.
// This method logs each invocation.
void swap(T& m, T& n) {
```

```
swap_count++;
swap_impl(m, n);
}
// Returns the number of comparisons this object has
// performed since it was created.
int comparisons() const {
return compare_count;
}
// Returns the number of swaps this object has
// performed since it was created.
int swaps() const {
return swap_count;
}
};
#endif
```

The **Comparer** objects in Code 10.5 swaps and compares any type of values that are meant to be used in an array or vector sorting program.

Conclusion

After taking you on a guide on ten of the must-know topics in C++, it is left to you now to develop your skills. And one thing with languages, be in English, French, or programming languages, practice makes perfect. The knowledge we've thought you in this book can be likened to learning the alphabet and some basic words in English. So, we'd advise you to practice with it more. Focus your mind on understanding why we use some statements; the result we want to obtain.

Go online, download all the necessary application that you find easy to work with, and if you'd have to pay for it, purchase it. And when you have everything all set and ready, pick up a project and learn. You could build your own little project, or you could pick an already made project and try to replicate it. If you ever get stuck anywhere, you can always go back, and view and the programmer made the project and proceeded. But always try to push yourself to your limit, think deep about it before checking.

It would also help you a great deal to pick up other really good books to get different perceptions about a section in C++ to broaden your understanding of it. Even though we can assure you of the quality of this book is sufficient to set you on the right path to becoming a success in C++, we would recommend you also read out other books on programming language.

Also, you could consider picking a different language other than C++ to learn side by side with C++. Leaning a new language is an efficient way to get a more in-depth understanding of C++. A new language brings in better comprehension of the programming language in general. It also broadens your mentality of solving problems, which brings out the specificities of the main C++ language. And even though you may not use this new language codes in C++, it would expose you to proven ideas that you can transpose into C++. A programming language we would advise you to study alongside C++ is Haskell. You can also study Java because it is way closer to our everyday English than C++, and it would be easier for you to understand. You can also check out our book on Java.

Additionally, always try to stay up to date. Catch up with the modern C++ features like C++11, C++14, C++ 17, and a host of other new features in the standard library. Some of its latest features, like lambdas, is easy to grasp. All you need is a good resource and time, and in no time, you'd see yourself becoming a master in them. And so, why not take a cup of tea, find a nice relaxing spot where you find it easier to focus and get started. And just like the famous saying goes, "a journey of a thousand miles begins with a step." So, take that step today, and let us be that guild that will set you on the right path.

C++

Simple and Effective Tips and Tricks to learn C++ Programming Effectively

BENJAMIN SMITH

Introduction

The main concern of this book is to explain as much of the fundamental concepts of C++ as possible. However, to refrain from overwhelming and discouraging the reader with unreasonably difficult concepts, the book only addresses those programming concepts that are very basic. Even in this beginner level's book, the reader will see a separation of concepts based on their level of difficulty. By keeping the most difficult parts of programming in the end, it allows the book more room for preparing the reader with the necessary information to understand the later topics. In this way, the book aims at providing the best learning experience for the readers and hopes that they build a solid conceptual foundation for learning intermediate and advanced level programming as well. The very first chapter of the book begins with a discussion focused on the very fundamentals of the C++ language itself, highlighting its characteristics, discussing traditional and object-oriented programming, and so on. After this chapter, the reader will have the necessary background knowledge to learn about functions, classes, objects, variables, types, and type conversions, etc. throughout the later stages of this book. However, it is recommended that the reader slow down the reading pace of the book as it seems a bit difficult at the start to understand. By going through the basic concepts detailed in the starting chapters and looking up any terms you don't understand on the internet, you will most likely succeed in becoming a fully-fledged beginner in C++ programming.

We also recommend the reader to go over the topic and the code a few times in case clarity of the concept is needed. The best way to ensure that you are fully able to understand the code is when you are writing and performing the actions simultaneously as it is being described. Other than that, we have ensured the proper coverage of the various topics that you will find in this book.

Chapter 1

The Fundamentals of C++

In this chapter, we will go through the very basics of the object-oriented programming language, C++. Knowing about the characteristics of a programming language is very important for users to consolidate new information when coding. As such, we will only go through the very important and commonly referred to characteristics of C++. In addition, this chapter will introduce the fundamental steps that are absolutely necessary to create a C++ program. To iterate over these outlined steps without sounding monotonous, we will use examples that will incorporate the information highlighted in these steps allowing the reader to retrace through the things they learned. Lastly, the main goal of this chapter is to make the reader familiar with the fundamental layout of a standard C++ program.

In this chapter, we will discuss three major topics,

1. The Fundamental Characteristics of C++.
2. Discussion of Object-Oriented Programming.
3. Translation and Creation of C++ Programs.

The Fundamental Characteristics of C++

It is important to understand that C++ is not a programming language that is purely focused on object-oriented programming. C++ is derived from the ‘C’ programming language and is a hybrid. Although C++ is different in some aspects from the original ‘C’ programming language, it still features the majority of the important functionalities found in the ‘C’ programming language. In other words, the C++ language also supports the features that are characteristic in the C programming language as well, such as:

- Modular programs that can be used universally.

- Efficient machine programming capability.
- Programs made in C++ can be easily ported over to other platforms.

Just as how C++ comes with all the important features of the C programming language, similarly, major contents of the code written in the C programming language can be reused in the C++ source code as well.

Another important characteristic of the C++ programming language is that it reinforces the concepts of C's object-oriented programming into itself. For instance, some object-oriented programming concepts are listed below:

- **Data abstraction**: creating classes to define objects while programming.
- **Data encapsulation:** obtaining a controlled access route to the data of the object.
- **Inheritance:** creating classes that are derived from other classes (classes can be even derived from multiple derived classes).
- **Polymorphism:** using an instruction set in such a way that it can boast different types of effect during the execution of the program code.

The C++ language features object-oriented characteristics of the C programming language and various elements from other programming languages. For instance, programming elements such as templates and exception handling offer incredible functionality when it comes to implementing your program efficiently. Not only that, these particular elements provide ease in your programming work while also ensuring that you have a clearer understanding of your programs.

Object-Oriented Programming

In this section, we will discuss and build a contrast between traditional procedural programming and object-oriented programming.

Traditional Procedural Programming

The main concept of the traditional procedural programming is based on separating the data which is supposed to be processed from the corresponding sub-routines and procedures (also known as data and functions). This significantly impacts the method through which a program handles data, for instance:

- It is the priority of the programmer to make sure that before the data can be used, it is initialized with proper and suitable values. Moreover, the programmer must also make sure that the data, and the proper values, can be passed to a function when required.

- The functions representing the data are specific. Due to this, if the representation of the data is changed (for example, if data is represented in the form of a record and that record is lengthened), the function which represents this data must also be modified accordingly.

Due to such features, this limits the productivity of the program and hinders the support for low program maintenance requirements. Moreover, the features of traditional procedural programming mentioned above make the program more prone to errors.

Object-Oriented Programming

In object-oriented programming, instead of emphasizing the elements of data and functions, the focus is primarily on the objects itself. In other words, the programmer gives importance to the elements of a program that highlight the main purpose of the program itself. For instance, in object-oriented programming, a program that has been created for the handling and maintenance of bank accounts will feature data objects such as interest rates, balance, credit and debit calculations, transfers, and so on. Such a program built with object-oriented programming will feature objects corresponding to each account in the program. Each of the objects represents properties and capacities that are crucial for the basic function of account management for the program.

In object-oriented programming, the properties and capacities correspond to data and functions, respectively, and these elements are then combined in the program. This is done by using classes. A class used in a program that is based on object-oriented programming will define a type of object by directly defining the properties and capacities of the object. Objects can also communicate with each other through sending messages. In this way, an object can activate another object’s capacities as well that serves multiple advantages along with ease in building the program.

Advantages of Object-Oriented Programming

In terms of software development, there are several advantages offered by object-oriented programming that make it more practical than traditional procedural programming. As traditional procedural programming lacks certain aspects that make it less likely to be used by programmers. Some of these advantages have been listed below:

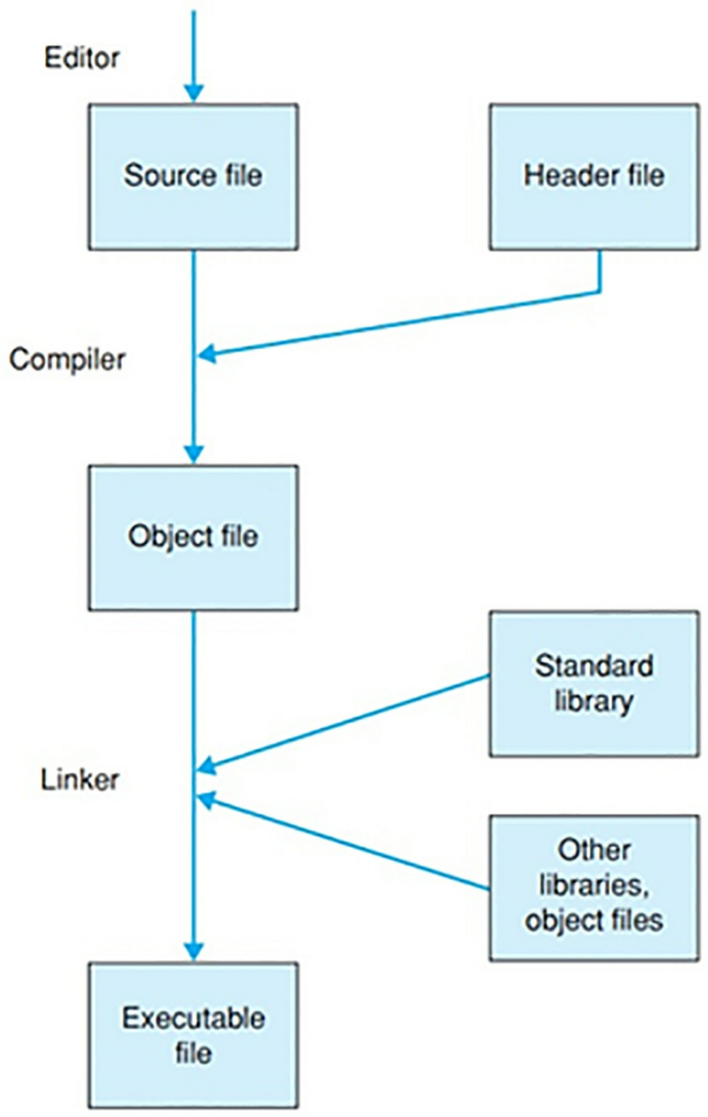
- **Less prone to errors:** an object defined in terms of its corresponding data can control the access attempts to this data. In other words, an object can reject error-prone access attempts to its data.
- **Ease of reuse:** objects that have been created are capable of maintaining themselves. Due to this characteristic of objects, this makes them easier to use in other programs as building blocks very conveniently.
- **Low maintenance needs:** According to the situation, an object is capable of modifying the representation of its internal data without needing to modify the application as well. This makes it more practical to use than traditional procedural programming.

Translating and Creating a C++ Program

Developing a C++ program is simpler than you might think. In this section, we will discuss how we can translate a C++ program and create a simple C++ program.

Translating a C++ Program

The following diagram depicts the process of C++ translation.



When translating a C++ program, three major steps are very important for the creation and translation process. These steps have been outlined below,

1. Saving the source code of the C++ program in a text file by using a text editor. The file which contains the source code of the program is known as the source file. The use of one source file is acceptable for short projects. However, for big programming projects, it is recommended to store the source code of the program in several source files so that they can be edited and translated separately with ease. This approach is known as **modular programming**.

2. Using a compiler to translate the program. The programmer feeds the compiler, a source file that contains the source code of the program which he wishes to translate. If the compiling process does not encounter any errors, then the output will be an object file that contains **machine code**. Such an object file is known as a **module**.

3. Using a **linker** to combine different modules to an object file to create an executable file. The modules that are being added to an object file through a linker contain functions that are either part of the program that has been compiled before-hand or contain functions from a standard library.

When creating a source file, it is important to note the extension which is being applied to the filename of the source file. Generally, the type of extension that is applied to the source file depends on the compiler being used to create the source file. The most commonly used file extensions are **.cpp** and **.cc**.

In some cases, the source file may require an extension file before it can be compiled. These files are known as **header** files. If a source file has corresponding header files available for the source code, then it becomes necessary to copy the contents of the header file into the source file before compiling it. Header files contain important data for the source file, such as defining the types, declaring the variables, and functions. Header files may or may not have a file extension. In cases where it does have an extension, then it is the **.h** file extension.

To perform the steps and tasks mentioned previously, programs use software known as a ‘compiler.’ Popular compilers nowadays offer an IDE along as well to get started with programming right away.

Creating a Simple C++ Program

We will now create a standard and simple C++ program. Afterward, we will proceed to discuss the coding elements used in this program.

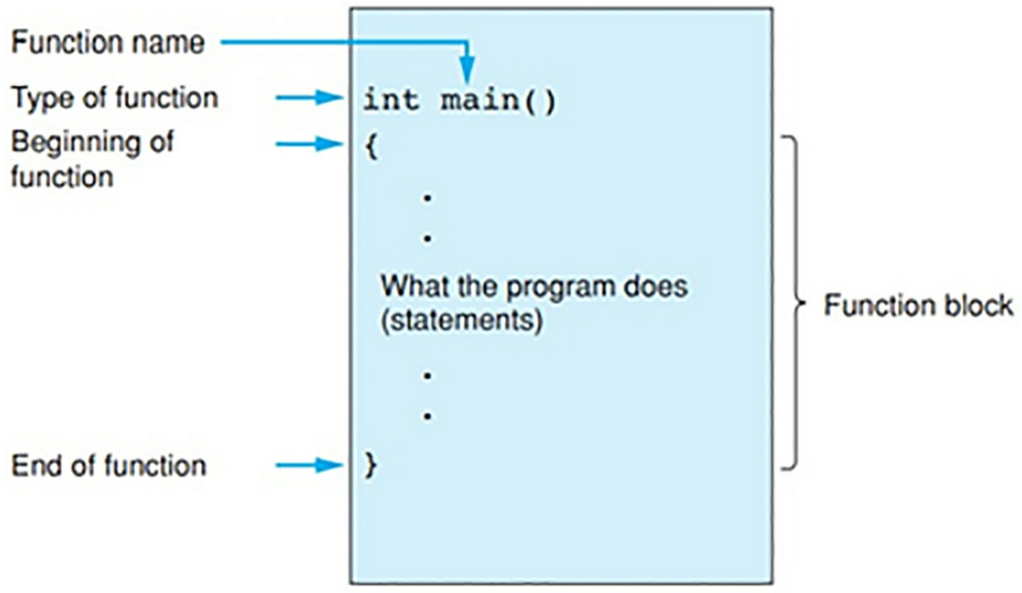
Here’s the code for the C++ program:

```
#include <iostream>
```

```
using namespace std;
int main()
{
cout << "Have a good day! << endl;
return 0;
}
```

The output of this program is a text; “Have a good day!”.

Before we delve into understanding the core elements of this program, here’s a visual representation showing the structural arrangement of the program to establish a basic understanding.



For simplification purposes, we can say that major parts of a basic C++ program are the objects and their corresponding ‘member functions and global functions.’ These objects and functions are not part of any class. Apart from completing the task for which it is made, a function also has the capability of calling other functions as well. Experienced programmers have the choice of either creating a function by themselves or use a function that is already available in the C++ standard library. For beginner’s it’s recommended to only use pre-built functions from the standard library as creating a personalized function may cause complications in the code that can be hard to work out. However, every programmer is required to create his own global function **main()**. This function is one of the main components of a program’s code.

The C++ program demonstrated above contains two of the most crucial elements of any C++ program, i.e., the **main()** function and the output string (the message that is displayed as the output of the program).

If you look at the first line of the code block, then you will see a hashtag symbol ‘#.’ This instructs the computer that this line of code is for the **preprocessor** step. You can think of **preprocessor** as a preliminary phase where no object is created yet, and code prepares for the upcoming instructions in the program. For instance, if you want the **preprocessor** to copy a file into the position where it is defined in the code of the source file, then you can use the ‘**#**’ symbol to do so. This has also been demonstrated below:

```
#include <filename>
```

In this way, we can include header files into the source code. By doing so, the program will be able to access the data in the header files. In the C++ program shown above, we can see in the very first line that the header file by the name of ‘iostream’ has been included in the source code through this method. The data contained by this file corresponds to the conventions that define input and output streams. In other words, the information in the program is considered as a stream of data.

The second line of code in the program refers to a namespace known as **std,** which contains predefined names in C++. To access the contents of this namespace, we use the **using** argument before defining the namespace.

The execution of the program itself begins from carrying out the instruction defined by the **main()** function. Hence, every C++ program needs to have the **main()** function to execute an instruction set. However, although the **main()** function is a global function that makes it different from other functions, the structure of how code is implemented with the function is exactly the same as any other typical function in C++.

In the C++ program demonstrated above, the **main()** function contains two statements,

1. cout << "Have a good day!" << endl;
2. return 0;

The first statement has two important components, **cout** and **endl**. Cout has been taken from the C++ standard namespace, and it represents 'console output.' Cout has been used to designate an object that will be handle outputting the text string defined in the statement. Moreover, the '<<' symbols represent the stream of the data in the program. These symbols tell the program that the characters in the string are supposed to flow to the output stream. In the end, **endl** indicates that the statement has finished and generates a line feed.

The second statement stops the execution of the **main()** function, and in this case, the program itself. This is done by returning a value of 0, which is an exit code to the calling program. It is always recommended to use the exit code 0 when highlighting that the program has been successfully terminated.

Creating a C++ Program that has Multiple Functions

Now that we know how a basic C++ program works and what core elements make up a simple program, let's proceed to understand a program that uses several functions instead of one. An example of such a C++ program has been shown below, also note that this program also has comments which are notes left by the programmer detailing what a specific line of code's purpose is. The comments are written after two back-slash symbols '//', and the compiler doesn't consider this as executable code and ignores it.

```
#include <iostream>
using namespace std;
void line(), message();                                    // Prototypes
int main()
{
    cout << "Hey! The program starts in main()."
         << endl;
    line();
    message();
    line();
    cout << "At the end of main()." << endl;
    return 0;
}
void line()                                            // To draw a
line.
{
    cout << "------------------------------" << endl;
}
void message()                                         // To display
a message.
{
     cout << "In function message()." << endl;
}
```

Now let’s execute this program and see what’s the output.

```
Hey! The program starts in main().
----------------------------------
In function message().
----------------------------------
At the end of main().
```

Through this example, we can understand the structure of a C++ program more intricately. Notice that unlike the simple program shown previously, which only had one function, this C++ program features several functions. However, C++ does not restrict the user to a defined order according to which the functions need to be defined. In other words, when working with a C++ program, you can define functions in any order that you may choose. For instance, you can define the **message()** function first, then the **line()** function, and finally, the **main()** function or in another order that you want.

However, it is recommended that you define the **main()** function first. This is because this function essentially controls the program flow. To understand this, we need to take a look at what the **main()** function primarily does. It calls those functions that haven’t been defined yet into the program where it’s being used. The **main()** function does this by providing the compiler with the **prototype** function. The **prototype** function contains all the necessary information for the compiler to help the **main()** function do its job.

In addition, if you take a look at this program, you’ll find some sentences that start after ‘//’ symbol. These are strings, and the program interprets them as comments. Comments are very useful when working on big projects that involve writing thousands of lines of code. By using comments, the programmer can essentially leave reminders as to what is the purpose of this line of code or an entire block of code, making it easier to debug, or for other users accessing the code to understand it’s structure. Moreover, comments also make it easier for programmers to make changes to the code later on. Hence, programmers practicing this habit often divert some hefty workload later on in the future.

Chapter 2

The Basic Data Types, Constants, and Variables Used in C++

In any programming language, being familiar with the generally used constants and variables is very important. In a programming language that is more object-oriented, such as C++, it becomes even more important to learn about at least the basics of the data types and objects that are commonly incorporated into C++ programs. As such, this chapter will focus on the elements mentioned above and aim to equip the reader with all the basics necessary for them to understand and create more complex and effective C++ programs in the future.

The Fundamental Data Types

In this section, we will discuss what a type refers to in a C++ program and understand the different fundamental data types available for use in the C++ programming language.

In programming, we have to take the different types of data into consideration, which can be used by the program to perform a task or solve a problem. Moreover, a computer does not work with only a single data processing method. To make use of the computer's ability to process and save data through various methods, we need to know what type of data we are dealing with or figure out the data type which is being fed into our C++ program.

Generally, there are four major categories of data types in C++. These four data type categories are:

- Boolean values
- Characters

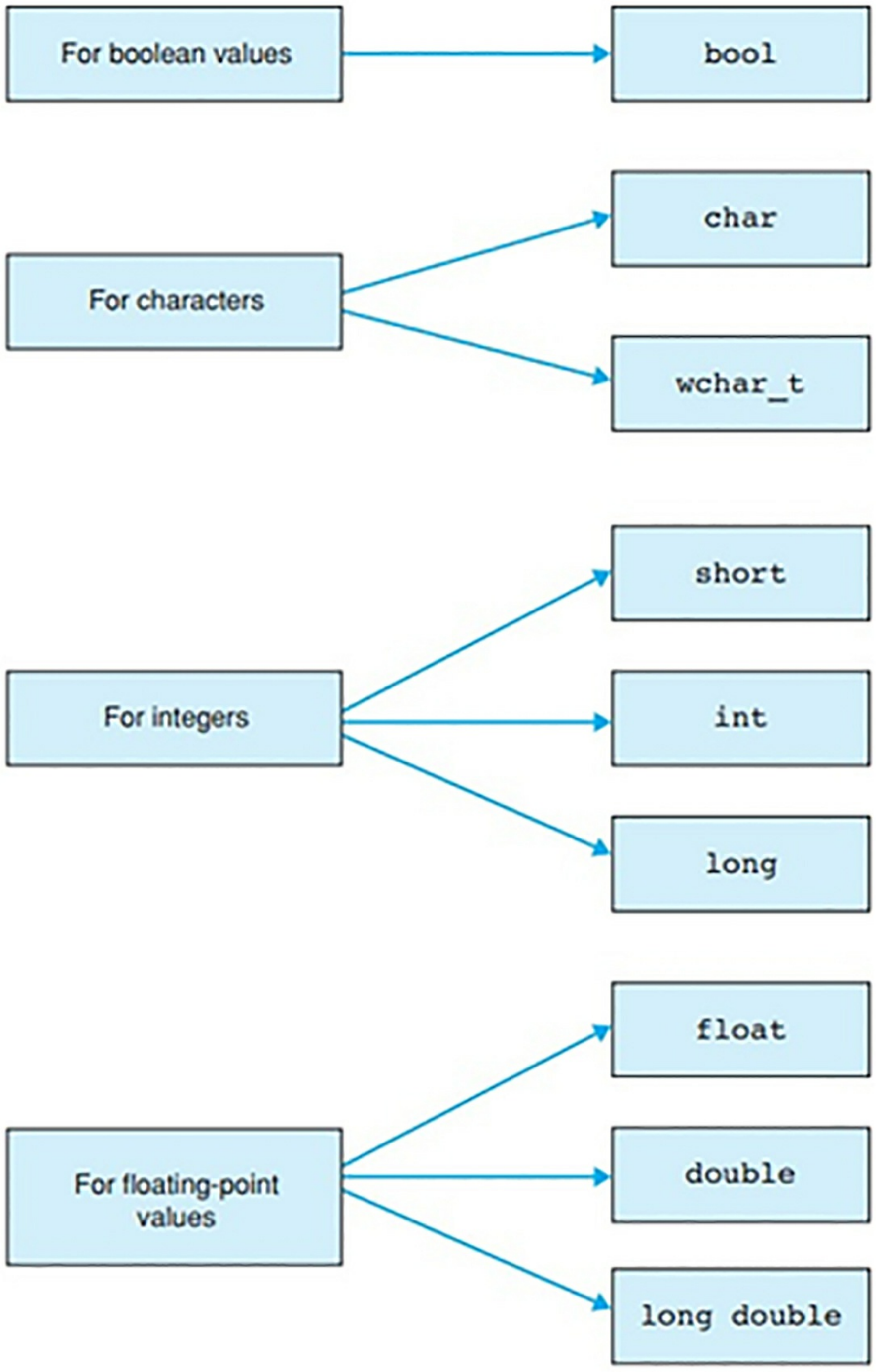
- Integers
- Floating-point values

Each data type requires a different approach to be adopted to make it possible for the computer to process the information they carry. That's why it is very important to identify the data type in the program. The purpose of a data type is to elaborate:

- The data's internal representation
- The memory value that needs to be allocated

For example, let's consider an integer that we want to store as data. The number -1024 needs only 2 or 4 bytes of memory as storage space. In other words, if we want to store such an integer, we will need to allocate 2 or 4 bytes to do so. To access this data, we will need to access that part of memory where it has been stored and to do this, we simply need to read the corresponding byte numbers. Moreover, the program through which we are accessing this data type must also interpret the sequence of bytes, representing the data in memory, as an integer with a negative sign.

Here's a chart representing the basic data types that are natively recognized by C++.



These are the fundamental data types that serve as the base for other data types such as vectors, pointers, and classes, etc. The C++ compiler has native support for these fundamental data types. As such, these data types are commonly referred to as **built-in types**. We will now proceed to discuss each category of data type individually.

The ‘bool’ Data Type

A value obtained as a result of a logical comparison or association by using logic gates (such as AND or gates) is known as a **Boolean** value and is referred to as a **bool** data type. The distinguishing feature of this value is that it can be either of one of the logical states, i.e., true or false. The internal value of these logical representations is ‘numerical value 1’ for the ‘true state’ and ‘numerical value 0’ for the ‘false state.’

The Character Data Types

The character data type is used to save the code, which refers to a specific character. To be more specific, in programming, each character code is represented by an integer that is associated with the character. For instance, the integer associated with the character code of **“A”** is ‘65’. Over the years, many character code sets have been developed to facilitate the digitization of writing. However, some sets have different character codes representing a particular character. So whenever a character is to be displayed on a screen connected to a computer, the associated character code is translated by the processor, and the resulting character is shown.

Although there are many character sets available in programming, the C++ language does not restrict the user to any particular set. However, character sets that contain the American Standard Code for Information Interchange, or more commonly referred to as ASCII code is preferred. This is a 7-bit character code set, and as such, it features 32 code characters and 96 printable characters.

In C++, the character data type features two major type containers, the **char type,** and the **wchar_t type**. The main difference between these two types is the storage they use to allocate the character codes. The **char type** uses 8-bits (or one byte) of storage, making it suitable for extended character sets such as the ANSI character set. At the same time, the **wchar_t type** uses 16-bits (or 2 bytes) of storage to allocate the character codes making it suitable for the modern Unicode characters. This is also the reason why the **whchar_t** type is referred to as the ‘wide character set.’

The Integer Data Type

There are three integral types used to represent integer data. These types are different from each other based on the range of values they can represent.

These integral data types are the following:

1. **int** and **unsigned int**
2. **short** and **unsigned short**
3. **long** and **unsigned long**

The table shown below highlights the properties of and specification (storage space and the supported range value) of each integral data type.

Type	**Size**	**Decimal Range of Values**
Char	1 byte	-128 to +127 or 0 to 255
Unsigned char	1 byte	0 to 255
Signed char	1 byte	-128 to +127
Int	2-byte resp. 4 byte	-32768 to +32767 resp -2147483648 to +2147483647
Unsigned int	2-byte resp. 4 byte	0 to 65535 resp. 0 to 4294967295
Short	2 byte	-32768 to +32767
Unsigned short	2 byte	0 to 65535
Long	4 byte	-2147483648 to +2147483647
Unsigned long	4 byte	0 to 4294967295

The **int** type is like a self-fitting and self-adjusting integral type. It adapts to the register of the computer on which it is being used. For example, if the **int** type is being used on a 16-bit system, then it will have the same specifications as the **short** type. Similarly, if the **int** type is being used on a 32-bit system, then it will have the same specifications as the **long** type.

You might have also noticed a data type that isn't a part of the integral data type family. This is the **char** type belonging to the character data type. The reason as to why the **char** type is included in this table is because C++ evaluates character codes just like any ordinary integer. Due to this, programmers can perform calculations on variables that are stored in **char** and **wchar_t** type in the same way as they would with variables stored in **int** type. However, it is important to keep in mind that the storage capacity of a **char** type is only a single byte. So, depending on how the C++ compiler interprets it (as either signed or unsigned), the range of values you can use with it are only from -128 to +127 and 0 to 255, respectively. On the other hand, the **wchar_t** type is an extended integral type as opposed to the **char** type being a simple **integral** type and is commonly defined as **unsigned short**.

Now let's discuss the **signed** and **unsigned** modifiers in the integral type. A signed integral has the highest valued bit as the representative of the sign of the integer value. On the other hand, unsigned integrals do not feature the highest bit representing the sign, and as a consequence, the values that can be represented by unsigned integral are changed. However, the memory space required to store the values is the same for signed and unsigned integral types. Normally, the **char** type is interpreted by the compiler as 'signed,' and as such, there are the modifiers result in three integral types, **char, signed char** and **unsigned char**.

Users can take advantage of a header file named as '**climits**' that defines the integral constants, for instance:

- **CHAR_MIN**, **CHAR_MAX**, **INT_MIN**, and **INT_MAX**.

As their name suggests, these constants represent the smallest and largest values of their corresponding integral types. To understand integral types better, let's implement this header file in a C++ program and use these constants to output the values of the **int** and **unsigned int** types.

```
#include <iostream>
#include <climits>                                  // Definition of
INT_MIN, ...
using namespace std;
int main()
{
   cout << "Range of types int and unsigned int"
        << endl << endl;
   cout << "Type                         Minimum
                                      Maximum"
        << endl
        << "----------------------------------------"
        << endl;
    cout << "int              "<< INT_MIN << "                "
                                 << INT_MAX << endl;
    cout << "unsigned int        " << "            0              "
                                << UINT_MAX << endl;
    return 0;
}
```

The Floating Point Data Type

Floating-point data refers to those numbers that have a fractional portion indicated by a decimal point. Unlike integers, floating-point values need to be stored according to a preset accuracy. This is in accordance with how decimal numbers are treated mathematically, i.e., the least significant numbers are ignored or rounded off. According to the preset accuracies, there are three floating-point types which have been listed below:

1. **float**: corresponds to a simple accuracy preset
2. **double**: corresponds to a double accuracy preset
3. **long double**: corresponds to a high accuracy preset

An important thing to note regarding floating-point types is that the maximum and minimum value range along with the accuracy of a particular type is dependent on two factors:

1. The total memory allocated by the floating-point type.
2. The internal representation of the floating-point type.

Now let’s discuss and clarify the concept of ‘Accuracy’ in floating-point numbers. In numerical representation, the more numbers a value has after decimal points, the more accurate it is. In other words, high accuracy means that the floating-point value has a higher amount of numbers coming after the decimal point. Similarly, low accuracy means that the floating-point value has fewer numbers after the decimal point. For example, the floating-point number 22.123456 has a higher accuracy as compared to the floating-point number 22.123.

Now we must understand how is this concept of ‘Accuracy’ leveraged by programmers. , when we have a number that is accurate up to six decimal places, we can store the same value as a separate and distinguishable number as long as it differs from the original number up to at least one decimal place. However, the same convention does not hold necessarily always hold true if we try to store a number that is accurate up to 5 decimal places along with the same number that is accurate up to 4 decimal places as separate numbers in a floating-type of 6 decimal place accuracy. To clarify, there’s no guarantee that the two numbers 22.12345 and 22.1234 will have different decimal places when their accuracy is brought up to 6 decimal places.

In cases where it becomes crucial for your program to represent a floating-point number in an accuracy that is strictly supported by a particular device or system, it is recommended to first refer to the values that have been defined in the header file named ‘**cfloat**.’

In conclusion, there are mathematical representations of data that are handled primarily handled by two data types. Depending on the arithmetic nature of the value (integer or a number with decimal points), it can be either stored in the **integral type** or the **floating-point type**. A recap of all the corresponding types has been listed below.

Integral Data Types

bool

char, signed char, unsigned char, wchar_t

short, unsigned short

int, unsigned int

long, unsigned long

Floating-Point Data Types

float

double

long double

The Institute of Electrical and Electronic Engineering, also referred to as IEEE, has provided the field of computer programming with a universal format that is used to represent the **floating-point types**. The following table briefly discusses this format representation.

Type	**Memory**	**Range of Values**	**Lowest Positive Value**	**Decimal Accuracy**
Float	4 bytes	–3.4E+38	1.2E—38	6 digits
Double	8 bytes	–1.7E+308	2.3E—308	15 digits
long double	10 bytes	–1.1E+4932	3.4E—4932	19 digits

In the following sections, we will briefly discuss the operator '**sizeof**' and talk about the classification of data types according to the nature of the object they represent.

The 'sizeof' Operator

When we need to confirm the required memory to store a certain type of object, we simply use the 'sizeof' operator for this purpose. For instance, if we use this operator as shown below:

sizeof (name of the object)

the program will tell us the memory necessary for storing the object. In other words, the operator 'sizeof' provides the object's memory requirements while the 'name' parameter defines the object itself, or it's the corresponding type. For instance, by inputting the proper data into the above parameter, let's say **sizeof(int)**. If you remember that the int type is adjusted itself according to the system, then you will also understand as to why the sizeof operator gives different values depending on the system (it can be either 2 or 4). Similarly, if we use this operator for floating-point type, i.e., **sizeof(float),** then we will be given a value of 4 representing the memory required to store the object.

Classification

Based on the nature of the object, we can classify two of the three fundamental data types (**integer types and floating-point types**) as arithmetic data types as we can perform arithmetic calculations on the variables of these types using arithmetic operators.

Until now, we have discussed objects that represented values and classified them accordingly. However, there are also those objects that do not represent any value, for example, a function. To classify such expressions, we simply represent them through the **void type**. In other words, the void type includes those expressions that do not represent any value. As such, a typical function call can be represented by a void type.

The Fundamental Constants

A constant is also referred to as a "**literal**." We deal with constants throughout programming, even if you don't know about them. For instance, we just learned about the Boolean data type. A Boolean value can be either 1 or 0, i.e., True or False. The keyword here 'True and False' are both Boolean constants. Similarly, every number, character, and even a string (sequence of characters) is a 'constant.'

Constants are directly related to data types. The very basic purpose of a constant is to represent values, which in turn represents the type. Hence, depending on how the constant is being used, we can define the type of the value accordingly.

Based on the above discussion, we can conclude that there are four fundamental constants in C++. These constants are:

1. Boolean constants
2. Numerical constants
3. Character constants
4. String constants

We will now discuss each fundamental constant separately.

1. Boolean Constants

A Boolean constant is a keyword used to represent either of the two possible values. True and False are both Boolean constants. Boolean constants belong to the **bool** type and can be used to set up conditionals within a program or even set flags that functions to represent the two states.

2. **Integral Constants**

Standard decimal, octal, or even hexadecimal numbers are referred to as integral constants. Let’s briefly discuss these three numbers and how they can be distinguished from each other.

- A decimal constant is a number belonging to the decimal number system (base 10). A decimal number never begins with a zero. 110, 124020 are both decimal numbers.
- An octal constant is a number belonging to the octal number system (base 8). An octal number always begins with a zero as the leading digit. For instance, 088 and 022445 are both octal numbers and are referred to as octal constants.
- A hexadecimal constant is a number belonging to the hexadecimal number system (base 16). A hexadecimal number always begins with a character pair. This character pair can be either “0x” or “0X”. For instance, 0x224A and 0X21b4F are both hexadecimal numbers. The Alphabetical numbers represent digits greater than nine up to 15 (for example, A represents 10, B represents 11 and F represents 15). There is no uppercase or lower-case capitalization restriction imposed on hexadecimal numbers.1

Usually, integral constants are assigned to the **int** type. However, if the constant value turns out to be too large for the **int** type to handle, then a type that is suitable to deal with that value will be used instead. Regardless, it is important to know the ranking of the integral types:

1. **int**
2. **long**
3. **unsigned long**

To designate either the long or unsigned long type to an integral constant, we simply attach the first alphabetical letter of the type to the number. For example, if we were to assign the number 15 to a long type, then we would do so by either adding ‘L’ or ‘l’ to the number. Similarly, if we were to assign the same number to the unsigned long type, then we would do so by using the **‘**UL’ or ‘ul’ letter. For assigning a number to the unsigned int type, we only need to use the letter ‘U’ or ‘u.’ This has been demonstrated below:

15L and 15l correspond to the type long

15U and 15u correspond to the type unsigned int

15UL and 15ul correspond to the type unsigned long

A detailed example of all the integral constants has been demonstrated in the table shown below.

Decimal	**Octal**	**Hexadecimal**	**Type**
16	020	0x10	int
255	0377	OXff	int
32767	077777	0x7FFF	int
32768U	0100000U	0x8000U	unsigned int
100000	0303240	0x186A0	int (32-bit) long (16-bit)
10L	012L	0xAL	long
27UL	033UL	0x1bUL	unsigned long
2147483648	020000000000	0x80000000	unsigned long

From the table shown above, you can see how each value has been represented in different ways.

Here’s an example of a program that incorporates the integral constant concepts we have discussed so far.

```
// To display hexadecimal integer literals and
// decimal integer literals.
//
```

```
#include <iostream>
using namespace std;
int main()
{
   // cout outputs integers as decimal integers:
   cout << "Value of 0xFF = " << 0xFF << " decimal"
        << endl;                    // Output: 255 decimal
   // The manipulator hex changes output to hexadecimal
   // format (dec changes to decimal format):
   cout << "Value of 27 = " << hex << 27 <<" hexadecimal"
        << endl;                    // Output: 1b hexadecimal
   return 0;
}
```

Floating-Point Constants

A floating-point value has two elements, an integral element, and a fractional element. The fractional part of the number is separated from the integer by a decimal point. That's why although floating-point values are usually represented as decimals, they can also be represented through exponential notation as well. For example, a typical floating-point number that is accurate up to one decimal place can be represented as shown below:

20.7

Similarly, if we have a number 1.5*10-^{2} then instead of inputting it in its decimal form, we can simply store it in its exponential form as well, which would be **1.8E-2**. The arithmetic type used to store these objects would be the **double** type by default. However, the constant can be manually designated as a **float** type as well by adding 'F' or 'f' to the value. Similarly, you can assign it to the **long double** type as well by adding 'L' or 'l.'

The following table shows a few examples of how floating-point constants can be represented in different ways.

7.19	16.	0.55	0.00001
0.719E1	16.0	.55	0.1e-4
0.0719e2	.16E+2	5.5e-1	.1E-4
719.0E-2	16e0	55E-2	1E-2

Character Constants

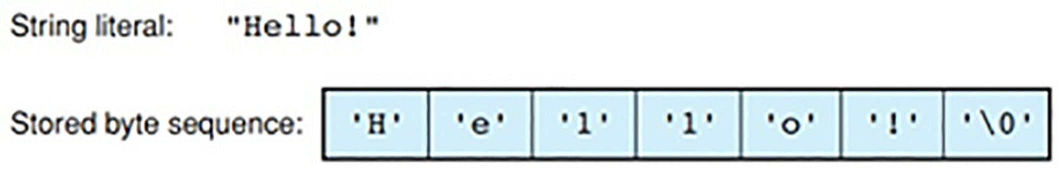
A character that has been enclosed in single quotes is identified as a character constant. Character constants are generally assigned to the **char** type. Each character has a numerical code that represents the character itself. For instance, in the ASCII code, the character ‘A’ is numerically represented by the number ‘65’. The table shown below elaborates a few character constants along with their decimal value in the ASCII code.

Character Constant	**Character**	**ASCII Code Decimal Value**
‘A’	Capital Alphabet A	65
‘a’	Small Alphabet a	97
‘ ’	Blank space	32
‘.’	Dot	46
‘0’	Numerical Digit 0	48
‘\0’	Terminating Null Character	0

String Constants

We are already familiar with string constants as we had dealt with them in the first chapter when we were discussing the text output for the **cout** stream in C++ programs. Regardless, you should remember that a sequence of characters (such as a sentence or a phrase) enclosed within double quotes is considered as a string constant. For example, “Press the Accept button to proceed!” is a string constant.

It is important to note that when a string sequence is stored internally, the double quotes are not included. Instead, a terminating null character (\0) is placed at the end of the string sequence to indicate that the sequence has ended. As such, apart from the bytes required to store the individual characters, a string sequence also uses another byte to store the terminating null character as well. That’s why string constants take up one additional byte than their usual memory requirements. The following figure depicts this internal representation of the string sequence “Hello!”



Escape Sequences

Another aspect to consider when talking about characters in programming is those that are non-graphic or not displayed on the screen but do have a purpose. For example, look at your keyboard and go through the buttons that, when pressed, display a character on a word processor. As soon as you think about it, you'll notice that certain characters do not display any character on the screen when pressed, such as the 'Tab' button, the 'Shift' button, and so on. Such non-graphic characters are referred to as 'escape sequences' and its effect depends on the device on which it is being used. For example, the Tab escape sequence (\t) depends on the default setting of the width space that has been predefined, which is eight blanks.

In addition, note that every escape sequence has a back-slash at the start of the character. The table shown below elaborates on several escape sequences along with their decimal values and effects.

Character	**Definition**	**ASCII Code Decimal Value**
\a	alert (BEL)	7
\b	backspace (BS)	8
\t	horizontal tab (HT)	9
\n	line feed (LF)	10
\v	vertical tab (VT)	11
\f	form feed (FF)	12
\r	carriage return (CR)	13
\”	" (double quote)	34
\’	' (single quote)	39
\?	? (question mark)	63

\\	\ (backslash)	92
\0	string terminating character	0
\ooo (up to 3 octal digits)	the numerical value of a character	ooo (octal)
\xhh (hexadecimal digits)	the numerical value of a character	hh (hexadecimal)

Octal and hexadecimal escape sequences allow users to represent character codes differently. For example, if you want to express the alphabet ‘A,’ then you can convert its decimal value to an octal value and then use the octal escape sequence to express it. In ASCII code, you will need to use the decimal value 65; however, if you’re not using the ASCII character code set, then you can simply use ‘\101’ to express the same alphabet. Similarly, you will have to use ‘\x41’ if you want in the hexadecimal escape sequence to express the alphabet ‘A.’ Although escape sequences can be used in this way, they are primarily used to express those characters that are non-printable. For example, if you want to represent the character ‘ESC,’ which serves as a control sequence for printers, then you can express it by using an escape sequence (for octal \33 and for hexadecimal \x1b).

The Fundamental Variables

Variables are commonly referred to as ‘objects,’ especially if they are part of a class. Variables are very important in programming as they serve as containers for data (numbers, characters, and even entire sequences). Moreover, by using variables, the program can easily process the data. In this section, we will discuss all the fundamental concepts about variables.

Defining Variables

A variable is an empty container that is abstract to the program until it has been defined. By defining a variable, we mean that we are storing data inside this abstract container by allocating memory to it, making it meaningful for the program. So, in a sense, we specify the data type and reserve the memory, which is to be used by the variable once it has been initialized. Once a variable is defined, think of it as a redirection tool. When a user refers to this variable, the program automatically fetches the corresponding data associated with the variable in the memory space in which it is allocated, so that the program can process it. In this way, we can perform operations on this data linked to the variable easily. A standard definition of a variable has a proper syntax that needs to be followed. This syntax is:

```
typ name1 [name2 ... ];
```

By looking at this syntax, we can see that the argument **‘typ’** is actually mentioning the variable’s type. ‘name1’ is simply the variable’s name that the programmer specifies. Once defined, whenever we want to specify the variable in any part of the program, we use this particular name. In this syntax, we can also see a square bracket that houses another variable by the name of ‘name2’. The square brackets tell the program that the variables mentioned inside it are not entirely necessary, and the program can ignore this portion if it needs to. This means that we can define and store multiple variables in one go. The following program demonstrates how we can define variables practically:

```
char c;
int i, counter;
double x, y, size;
```

Another important thing to note is that variables can be defined in two ways. They can be defined within the function of the program or outside the program’s function. For example, at any portion of the program as long as it isn’t done within the function. However, defining a variable either within or outside a function has certain effects, respectively. These have been listed below:

1. A variable defined outside a function has a global property within the program. This means that all of the program’s functions can use this variable.

2. A variable defined inside a function has a local property within the program. This means that the variable can only be used by the function in which it has been originally defined and not by any other function of the program.

A local variable can be defined at any point in the function where a statement is permissible.

Initialization

The initialization of a variable essentially refers to the assigning of value to the variable mentioned above. When a variable is being defined, it is also initialized at the same time. Once the variable is defined, we simply assign a value to it immediately after the definition syntax to initialize it.

A variable can be initialized by using either of the two methods

- After defining the variable, putting an equal sign (=) right next to it and inputting the desired value. For example:

char k = ‘a’;

- Enclosing the desired value of the variable in round brackets immediately after defining the variable. For example:

char k(a); Similarly **float x(1.112);**

A global variable that is not initialized has a default value of ‘0’. However, this does not apply to local variables as a local variable that hasn’t been initialized will be assigned an undefined value by default.

Here’s an example of a program incorporating the concepts we have discussed so far.

```
// Definition and use of variables
#include <iostream>
using namespace std;
int gVar1;                                    // Global variables,
int gVar2 = 2;                               // explicit initialization
int main()
{
  char ch('A');                              // Local variable being
initialized
                                  // or: char ch = 'A';
  cout << "Value of gVar1:           " << gVar1 << endl;
  cout << "Value of gVar2:      " << gVar2 << endl;
  cout << "Character in ch:     " << ch << endl;
  int sum, number = 3;                    // Local variables with
                                            // and without initialization
  sum = number + 5;
  cout << "Value of sum:               " << sum << endl;
return 0;
}
```

Upon executing this program, the computer will display a result as shown below:

```
Value of gVar1: 0
Value of gVar2: 2
Character in ch: A
Value of sum: 8
```

Constant and Volatile Objects

In this section, we will discuss two important keywords in programming that can heavily affect the properties of an object. These keywords are:

- **const** (For constant objects)
- **volatile** (For volatile objects)

Constant Objects

Let's say we already have an object of a specific type available for use. We can change the properties of the object by using the **const** keyword in place of its type. Doing so will create an entirely new object that is specifically 'read-only.' As we know from our daily interaction with computers, a read-only file can only be accessed and cannot be modified by the user. The same is the case for a constant object. Since its properties have become read-only (in other words, the object itself has become constant), it renders the program helpless when it attempts to modify the object. So a constant object, once defined and initialized, cannot be modified later on. Due to this nature, the programmer must initialize the constant object at the same time it is defined; otherwise, it will be a read-only object that is assigned a default value.

An example of a constant object being defined and initialized is shown below:

```
const double pi = 3.1415947;
```

By creating such an object, the program will not be able to modify the value of this object in any scenario. Even if we introduce a statement like this into the program:

```
pi = pi + 2.0;      // invalid
```

The program will only return an error message as a result.

Volatile Objects

A volatile object is the polar opposite of a constant object. However, volatile objects are rarely used in programs. An object created by using the keyword **volatile** will have properties that allow not only the program in which it resides but also other programs and external events as well to modify the object. External events are usually triggered through interrupts or hardware clocks, for instance:

```
volatile unsigned long clock_ticks;
```

In this way, even if the program does not directly modify the values of this variable, the hardware clock definitely will. As such, the program has to assume the value of this variable has changed since the last time it was accessed. For this purpose, the compiler constructs a machine code that allows it to scan and access the variable's current value instead of outputting the value on which it was initialized.

Another interesting point is that we can use both **const** and **volatile** keywords at the same time on the same variable. For example, if we were to use these keywords, then the resulting variable's properties cannot be modified by the program itself. Still, it can be modified by external events such as the hardware clock. An implementation of this concept has been shown below:

```
volatile const unsigned time_to_live;
```

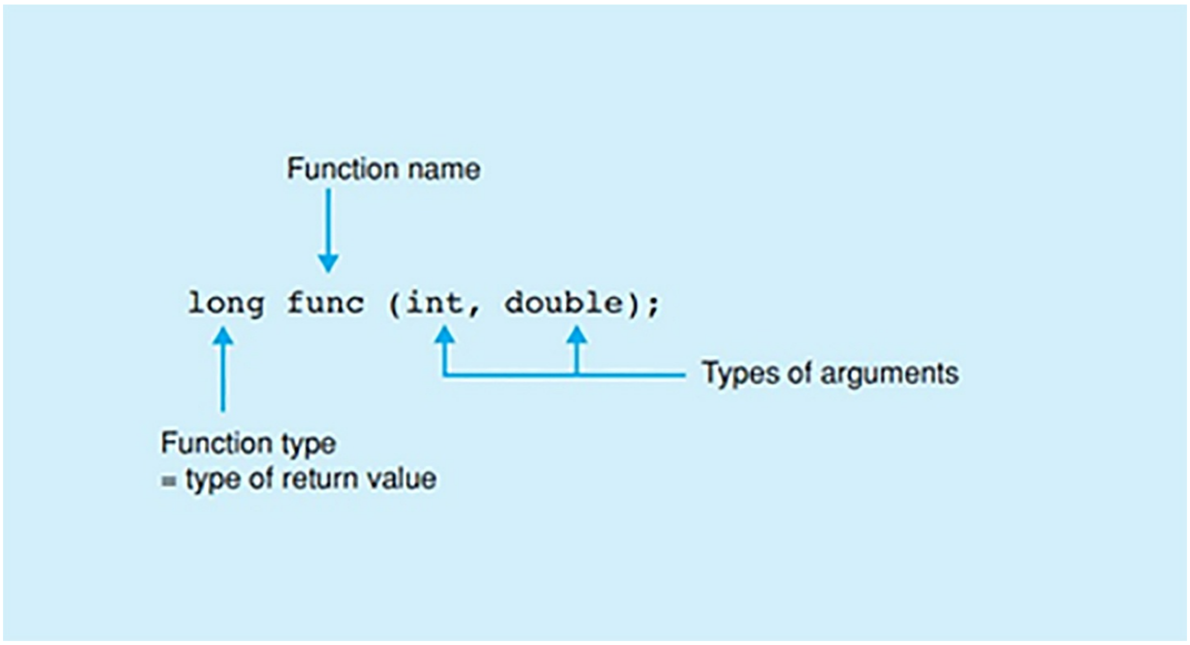
Chapter 3

Functions and Classes in C++

In this chapter, we will be taking another important step towards exploring the realm of C++ programming. This chapter builds upon some of the fundamentals that were briefly discussed or displayed in the first chapter's C++ program. Functions and classes are integral parts of a program, and knowing how to use them ensures that the programmer's skills have a strong foundation.

In this chapter, we will learn how to declare functions in a program and call a function in a program. In addition, readers will also be instructed on how to properly use standard classes in C++ along with header files and string objects that belong to the **string** class.

Before we go into theoretical details and discussion describing the basics of declaring functions in a program, let's first see a demonstration and build the concept from that. The prototype shown below depicts the structure of a typical function that has been declared:



If we analyze this function, we can see that the statement provides the following information to the C++ compiler:

- The name of the function to be used is **func**
- We are calling upon this specific function through two arguments. Both of these arguments are **type** arguments with the first one being **int** and the second one being **double**
- The value that the function will return should be a **long** type.

The following table highlights some of the most common and standard mathematical functions available for use in C++.

double sin (double);	//Sine
double cos (double);	//Cosine
double tan (double);	//Tangent
double atan (double);	//Arc Tangent
double cosh (double);	//Hyperbolic Cosine
double sqrt (double);	//Square Root
double pow (double, double);	//Power

double exp (double);	//Exponential Function
double log (double);	//Natural Logarithm
double log10 (double);	//Base-ten Logarithm

Declaring Functions

When writing a program, the compiler is like the main player who will execute the instruction set you are writing down. Following this analogy, it only makes sense that if we tell the compiler about a function that it does not know, it will not be able to execute it. Hence, a programmer needs to declare the functions and names (apart from keywords) for variables and objects; otherwise, the compiler will simply return an error message. This is why declaring functions is very important before we can even use them in our program. Usually, the point where you define a function, or a variable is also the point where they are declared as well. However, not all functions need to be defined for them to be declared. For example, if the function you wish to use has already been defined in a library, then you only need to declare that function instead of defining it again in your program.

Like a variable, a function features a name along with a type. The type of value defines the type of function it is supposed to return. In other words, if a function is supposed to return a **long** type value, then the type of the function will also be '**long.'** Similarly, the argument's type also needs to be properly specified for the function to work properly. In other words, to declare a function, we need to provide the compiler with the following information:

- The function's name and type
- Each argument's type

Such a structure makes up a function prototype. An example of a function being declared has been shown below

```
int toupper(int);
double pow(double, double);
```

The following statement tells the compiler that the name of the function is

toupper(), and its type is **int**. Since it is an **int** type function, the value it will return will also be an **int** type. The argument's type is also an **int,** so the function will expect an input argument that will be of the **int** type.

Similarly, the second function's name is **pow(),** and its type is **double**. It has two arguments, and both are of the type **double,** meaning that when this function is called, then it needs two arguments of the type mentioned above to be passed to it. We can follow up such types of function prototypes with names as well, but the names will be considered as comments by the compiler.

Now, let's discuss another case where a function has already been declared in the header file included in our program. For instance the function prototype:

```
int toupper(int c);
double pow(double base, double exponent);
```

It is the same for the compiler as the function prototype that has been demonstrated previously. If this function has already been declared in the header file (which has been added to the program initially using the directive #include), then we can use the function immediately without declaring it. For example, if we include the C++ math library 'cmath,' then we can use the standard mathematical functions immediately. The following program demonstrates the inclusion of this library to use the mathematical functions immediately.

```
// Calculating powers with
// the standard function pow()
#include <iostream>        // Declaration of cout
#include <cmath>           // Prototype of pow(), thus:
                           // double pow( double, double);
using namespace std;
int main()
{
  double x = 2.5, y;
  // By means of a prototype, the compiler generates
  // the correct call or an error message!
  // Computes x raised to the power 3:
  y = pow("x", 3.0);       // Error! String is not a number
```

```
    y = pow(x + 3.0);        // Error! Just one argument
    y = pow(x, 3.0);         // ok!
    y = pow(x, 3);           // ok! The compiler converts the
                             // int value 3 to double.
    cout << "2.5 raised to 3 yields: "
         << y << endl;
    // Calculating with pow() is possible:
    cout << "2 + (5 raised to the power 2.5) yields: "
        << 2.0 + pow(5.0, x) << endl;
    return 0;
  }
```

The output of this program will be

```
  2.5 raised to the power 3 yields: 15.625
  2 + (5 raised to the power 2.5) yields: 57.9017
```

Function Calls

A function call is leveraging the properties and results of a function and passing it on to the variable calling the function mentioned above. For example,

y = pow(x, 2.0);

In this example, the variable ‘y’ is calling upon the **pow()** function. The result of the function is $x^{2,}$ and this exponential power is then assigned to the variable ‘y’ calling the **pow()** function. In simpler terms, a function call represents a value. Since we are simply representing a value, we can also proceed to apply some other mathematical operations in a function call as well, like addition, subtraction, and multiplication, etc. For instance, we can perform an addition calculation on a function call for **double** values as shown below:

cout << 2.0 + pow(5.0, x);

In this statement, the value of ‘2.0’ is added to the value returned by the **pow()** function. Afterward, the **cout** argument displays this result as the final output of the statement.

It is important to remember that, even though any type of expression, be it a constant or mathematical expression, can be passed to a function as the argument, we must make sure that the argument is passed on is of a type that the function is expecting. In other words, the type of argument being passed should be in accordance with the type that was specified when the function was initially defined.

To ensure that the argument's type is indeed correct, the compiler refers to the prototype function to double-check the type that has been specified in the input argument. In the scenario where the compiler identifies that the type that has been initially defined in the function prototype does correspond to the argument's type being inputted, the compiler tries to convert it to the desired type. But this is not always possible, so the type conversion can only take place if it is feasible. This has been demonstrated in the code of the program shown previously as:

```
y = pow( x, 3); // also ok!
```

Note that the compiler expects a **double** type value instead of an **int** type value (3). Since the argument's type does not correspond to the prototype function, the compiler performs a type conversion of **int** to **double**. However, if the number of arguments being input is more or less than specified by the function prototype or the compiler cannot perform type conversion. It will simply return an error message. At this point, you can easily point out the origin of the error and fix it in the program's developmental stages, saving you from dealing with runtime errors.

In the following example:

```
float x = pow(3.0 + 4.7); // Error!
```

The compiler identifies that the number of arguments being used does not comply with the structure specified in the function prototype. Moreover, the return value of the function is a **double** type, and it cannot be assigned to a variable of a **float** type, hence making the type conversion impossible for the compiler. In this case, the compiler will simply generate an error.

Functions Without Return Values or Arguments

In most cases, the purpose of a function call would be to use it returns for the variable calling the function. However, we can also create a function that executes a task but does not necessarily return any particular value to the variable calling upon this function. For such purposes, we use the **void** type with these functions. In other programming languages, this is commonly referred to as a procedure.

To demonstrate the usability of a function call without a return value, let’s first consult the statement shown below:

```
void srand( unsigned int seed );
```

In this example, we see a function **srand()** being used to generate a random number. The function does this by initializing an algorithm that produces the random number for the output of the function. Since this function does not return a tangible value, we assign it the **void** type. The arguments of this function show that we are using an **unsigned int** value, which is to be passed to the function for use. This value acts as a seed for the random numbers that the function will produce, enabling it to generate a series of random numbers.

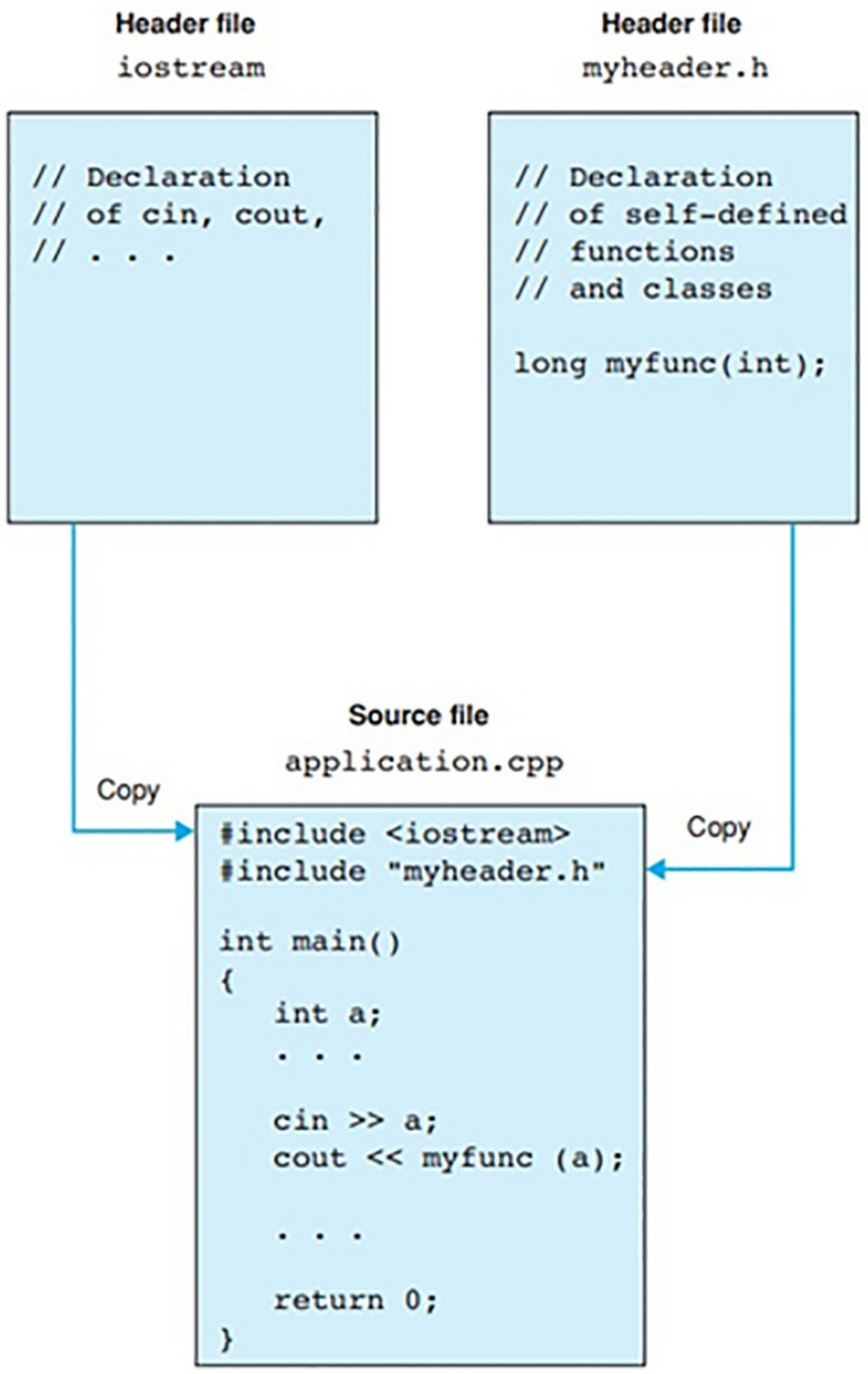
Now that we have discussed a function that can perform an action without returning a value let’s discuss a function that does not expect an argument. If we have defined a function without specifying any arguments, then the programmer must declare this prototype function as **void**. Alternatively, we can also leave the argument space empty in the parenthesis. This has been demonstrated below:

```
int rand( void );     // or int rand();
```

The program will call the function **rand()** without providing it with any arguments. Since no arguments have been specified, the function simply generates a random number between 0 and 32767 and returns it as the output value. In this way, we can generate a series of random numbers by repeatedly calling this function.

Header Files

, header files are those text file components in a program module that feature declared functions and macros. For a program to make use of the data contained within header files, they need to be added into the program’s source code by using the **#include** directive. A basic chart elaborating on how to use header files has been shown below:



```
// Declaration
// of cin, cout,
// . . .
```

```
// Declaration
// of self-defined
// functions
// and classes

long myfunc(int);
```

```
#include <iostream>
#include "myheader.h"

int main()
{
   int a;
   . . .

   cin >> a;
   cout << myfunc (a);

   . . .

   return 0;
}
```

By understanding this diagram, we can extract the following points of important information regarding how to use header files:

- Header files containing the declared functions and macros being used in the program should always be included at the beginning of the program's source code. If not, the compiler will not be able to refer to the function prototypes when executing the functions in the program itself.

- Multiple header files cannot be included with a single **#include** directive. If we want to add more than one header file into the program, then we will have to use separate **#include** directives to do so, as shown in the diagram.
- The name of the header file should be properly written (taking note of the upper and lower case along with any punctuation) when being declared by the **#include** directive. The header file's name can be either enclosed in angled brackets "<>" or in double-quotes, as shown in the diagram.

Searching for Header Files

Usually, whenever you include a header file, it gets automatically stored in a separate folder in your files directory. The directory where the header file that has been added to the program is stored is known as **'include**.' If you want to make changes to the header file or want to access it manually to use it with another project, then you first need to refer to the method you used to include it into the current project. If you have used angle brackets (<>) to include the header file, then it will be in the **include** directory. However, since most of the programmers using C++ create their own header files, they usually store it in the same folder where the project is stored. In such cases where the header file is not in the **include** directory, then you will need to add it to the source code file using double quotes (""). Similarly, if a header file has been included in the program's source code using double quotes, then you need to search the project folder itself to find the header file.

Standard Class Definitions in Header Files

By now, we know that a header file contains important data such as function prototypes that help the programmer to code functions with ease efficiently. However, a header file is not that one-dimensional. It can store other important programming data as well such, as one which we will discuss in this section, standard classes that have already been defined. Programmers can include objects and define classes inside the header file and use them in their program just as how they would use function prototypes. So, any class and object that has been defined in the header file, the program can access and use this data to execute tasks.

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

In this example, you can see two statements. Our concern is with the first statement only, i.e., the inclusion of the **istream** and **ostream** classes in the program. By adding the header file containing these classes, the program becomes capable of using the **cin** and **cout** streams. This is because **cin** is an object that belongs to the **istream** class, and similarly, **cout** is an object that belongs to the **ostream** class.

Using Standard Header Files

The table shown below highlights some of the header files that include standard C++ libraries.

algorithm	**ios**	**map**	**stack**
bitset	**iosfwd**	**memory**	**stdexcept**
iomanip	**locale**	**ssstream**	**vector**
complex	**iostream**	**new**	**streambuf**
functional	**list**	**set**	**valarray**
dequeue	**istream**	**numeric**	**string**
fstream	**limits**	**queue**	**utility**
exception	**iterator**	**ostream**	**typeinfo**

Since C++ can use most of the libraries used by standard C, here's a list of header files that include C standard libraries.

assert.h	**limits.h**	**stdarg.h**	**time.h**
ctype.h	**locale.h**	**stddef.h**	**wchar.h**
errno.h	**math.h**	**stdio.h**	**wctype.h**
float.h	**setjmp.h**	**stdlib.h**	**ios646.h**
signal.h	**string.h**		

At first glance, we can see that the header files shown in the two tables have one key difference, and that is the inclusion of the **.h** extension. This is because, in C programming language, header files are indicated by the extension '**.h**.' However, this is not the case for C++ header files as their declarations are all contained within their own namespace (**std)**. However, it is important to specify to the program that you will be using the **std** namespace globally to refer to identifiers. Otherwise, the compiler will not be able to recognize the **cin** and **cout** streams without the **using** directive.

```
#include <iostream>
```

In this case, the compiler cannot identify the **cin** and **cout** streams.

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

By adding the **using namespace std** directive, the program can now easily identify and use the **cin** and **cout** streams without the need for any syntax. Note that this demonstration also added the **string** header file as well. This will allow the program to perform string manipulation tasks in C++ easily.

C Programming Language Header Files

The C++ programming language has adopted and incorporated all of the standard libraries of the C programming language, making them available for use by C++ programs. As such, header files are also no exception. The header files which were originally standardized for the C programming language have been adopted for use by the C++ programming language as well. For example, we can use the C programming language's standard math library in C++ as well by using the following statement as shown below:

```
#include <math.h>
```

Although we can use standard C libraries, there is one complication that can cause quite a problem. When using a C header file, the identifiers declared in this file become globally visible. This can cause complications in big programming projects. To take care of this issue, we simply use another header file in addition to the C header file in C++. This additional header file declares these identifiers in the **std** namespace, thus solving the issue of identifier conflicts. For example, if we are using **math.h** library in C++, then we will accompany it with another header file named **cmath**. This **cmath** library features all of the data declared in the **math.h** library, but the difference is that the identifiers in **cmath** have been declared in the **std** namespace. This has been demonstrated below:

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

An important topic to clarify while we are talking about header files is that the **string** header file and **string.h** or **cstring** do not offer the same functionality. The **string** header file only defines the **string** class while the **string.h** or **cstring** header files declare those functions and variables that are used to manipulate string data in C programming language. In simpler terms, the **string.h** and **cstring** header files allow the user to access the functionality of the C string library, while the **string** header file's only purpose is to define the **string** class.

Here's an example showing the use of two header files, i.e., **iostream** and **string** in a C++ program.

```
// To use strings.
#include <iostream>       // Declaration of cin, cout
#include <string>         // Declaration of class string
using namespace std;
int main()
{
  // Defines four strings:
  string prompt("What is your name: "),
        name,             // An empty
        line( 40, '-'),        // string with 40 '-'
        total = "Hello ";      // is possible!
  cout << prompt;          // Request for input.
  getline( cin, name);       // Inputs a name in one line
  total = total + name;      // Concatenates and
                             // assigns strings.
  cout << line << endl       // Outputs line and name
       << total << endl;
  cout << " Your name is " // Outputs length
        << name.length() << " characters long!" << endl;
  cout << line << endl;
  return 0;
}
```

This example prints out the string using **cout** and **<<** statements. This program asks the user to input their name and then proceeds to display their input name along with the total number of characters contained within their name.

```
What is your name: Zoldyck Killua
--------------------------------------
Hello Zoldyck Killua
Your name is 13 characters long!
--------------------------------------
```

Using Classes in C++

The program shown in the above example can be seen to incorporate the **string** class into its core functionality. In the standard library of C++, multiple classes have been defined before-hand. These classes are very important for the foundations of a C++ program. They include the stream classes (**iostream**) and classes that effectively help represent strings for the program and conditions through which an error arises.

Even though there are many classes available for use, each class is unique with regards to their respective type that represents their properties and capacities. Although each class is unique, whether they are predefined classes or classes that have custom designed by the programmer, their properties and capacities are still governed by two main aspects.

- The data members of a class are responsible for defining the properties of their corresponding class.
- The class's **methods** (functions belonging to a class. These functions coordinate with the members of the class to carry out an operation) define the capacity of a class. The method of a class is also commonly known as its member function.

Creating Objects of a Class

Objects are variables that are assigned the class type itself. For instance, if we create an object for a string class, then the variable will have the **string** type accompanying it. This has been demonstrated in the example shown below:

```
string s("I am an object of the string class");
```

Whenever an object is created, it is allocated memory for its data members and then initialized with its corresponding values. In the example shown above, you can see that the variable **s** is of the **string** class type. An Object is also termed as an **instance** of the corresponding class it represents. Take the statement shown above as an example. The object **s** can also be referred to as an instance of the **string** class. The object is followed by a string constant (I am an object of the string class) and is ultimately defined and initialized in this way.

Calling Methods

In a class, all the methods that have been defined as ‘**public’** can be called by an object of this particular class. We must not confuse calling a public method with calling a global function. Although both share some basic similarities, however, there is one aspect that acts as a major differentiating factor between methods and functions. This aspect is that unlike **global** functions that can be used by multiple statements of the program in which it is defined, a **public** method can only be called for one particular object at a time only. A typical set up of a method and an object is shown below:

```
        s.length();          // object.method();
```

In this statement, the **length()** is the method for the object **s**. The main purpose of this method is to provide information regarding the total number of characters in the string. This has already been demonstrated in the program shown in the previous section. Instead of the **s** object, we can see that the **length()** method is being used with the **name** object.

Global Functions and Classes

A function that has been defined **globally** can be used by some classes as well. A global function being used by a class mainly executes certain operations for the class’s objects, which have been passed as arguments to the function. For example, consider the following statement, which features a global function **getline().** This has been used with an object **s** (which has been passed to the function as an argument):

```
getline(cin, s);
```

The global function executes an operation to store a line of keyboard input as a string. In this way, by pressing the return key, a new line character will be created by the **‘\n’** escape character, but this line will not be stored in a string.

Chapter 4

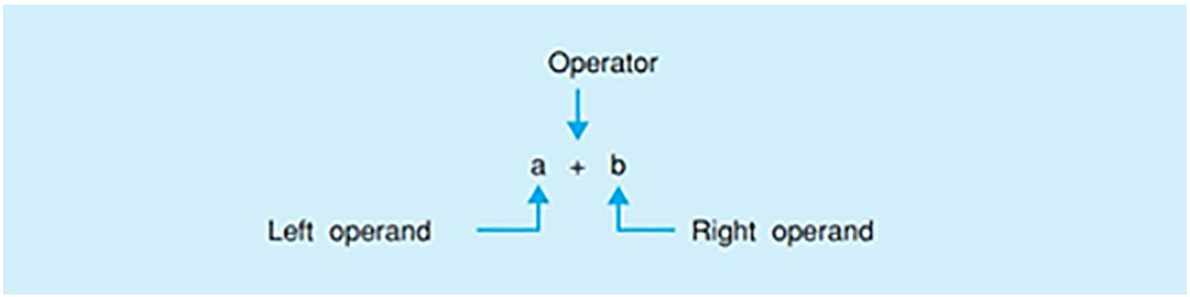
Operators For Fundamental Types

In this chapter, we will introduce those operators that are essential for performing calculations as well as selections for fundamental data types. Operators are similar to functions, but their syntax is quite different. Operators tools for performing specific operations such as arithmetic operations, comparison operations and logical operations, etc. We will be discussing some of the most basic operators used for fundamental programming.

Binary Arithmetic Operators

To make a program capable of processing the data input, the operations required for the process need to be defined. The execution of selective operations depends on the type of data being processed, such as adding, multiplying, or comparing numbers. However, multiplying strings would not be logical.

The most important operators, including unary and binary operators that are used for arithmetic processing, are discussed in the following section. Only one operand is used in unary operators while two are specified for the binary operators. Here's a diagram demonstrating a typical binary operator and operands.



For further conceptual clarification, a table listing the fundamental binary operators has been outlined below:

Operator	**Significance**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Here's a program demonstrating the use of binary operators:

```
#include <iostream>
using namespace std;
int main()
{
   double x, y;
   cout << "\nEnter two floating-point values: ";
   cin >> x >> y;
   cout << "The average of the two numbers is: "
        << (x + y)/2.0 << endl;
   return 0;
}
```

When executing this program, we will be given the following result:

```
Enter two floating-point values: 4.75 12.3456

The average of the two numbers is: 8.5478
```

The system can perform calculations with the help of arithmetic operators.

- In case of using integral operands to perform divisions, integral results will be produced. For instance, the computation of 7/2 will result in 3. Furthermore, one or more than one floating-point number operands will produce a result in the form of a floating-point number. For example, if 7.0/2 is calculated, the exact result will be 3.5.

- Integral Operands are the only ones that can make use of the remainder division, which is used to return the remainder of an integral division, e.g., 7%2 would produce the result 1.

Expressions

Simple expressions consist of only one constant, variable, or a function call, but when they are used as operands of operators, they form complex expressions. As such, an expression is usually a combination of operators and operands.

When utilized, all of the expressions return values except for those that feature a **void** type. The operands define the expressions in arithmetic calculations.

Examples: int a(4); double x(7.9);

```
a * 512       // Type int
1.0 + sin(x)  // Type double
x – 3         // Type double, since one
              // operand is of type double
```

If an expression is correctly used, it can act as an operand in another expression as well.

Example: 2 + 7 * 3   // Adds 2 and 21

When an expression is evaluated, the mathematical rules of multiplication before addition apply. So, the operators *, /, % are given priority to + and -. In the example provided above, the calculation 7*3 is performed first, and then two is added to it. On the other hand, the precedence order can be changed by inserting parentheses to the calculation that should be performed first.

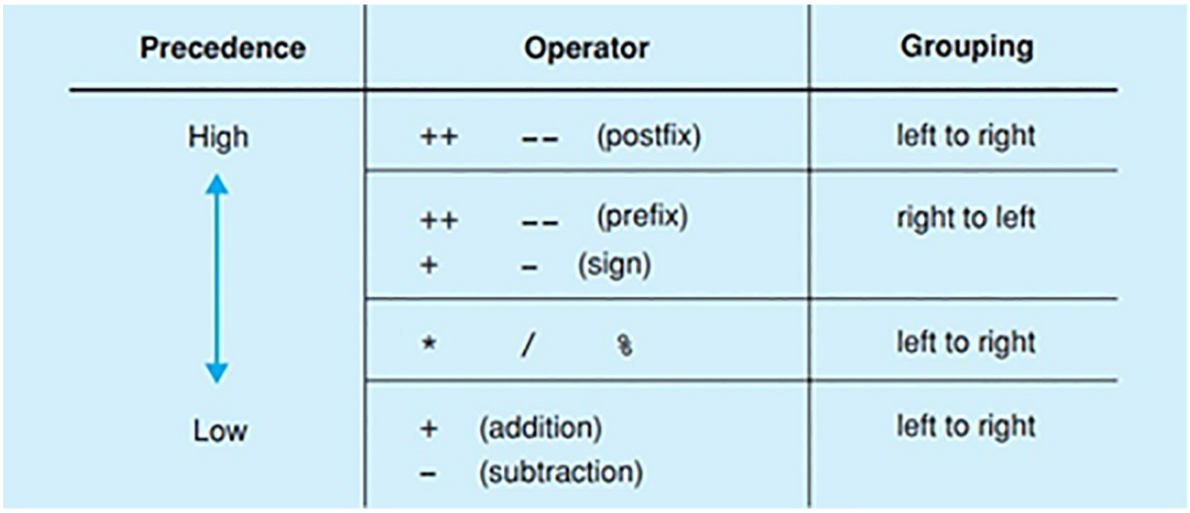
Example: (2 + 7) * 3   // Multiplies 9 by 3

Unary Arithmetic Operators

The total number of unary arithmetic operators is four:

- The Unary sign operator +
- The Unary sign operator –
- The increment operator ++
- The decrement operator –

However, not all of the unary arithmetic operators feature the same level of precedence. The precedence of the arithmetic operators has been demonstrated in the table below:



Precedence	Operator	Grouping
High	++ -- (postfix)	left to right
	++ -- (prefix) + - (sign)	right to left
	* / %	left to right
Low	+ (addition) - (subtraction)	left to right

The effects of the prefix and suffix notation of arithmetic operators have been demonstrated in the following sample program:

```
#include <iostream>
using namespace std;
int main()
{
   int i(2), j(8);
   cout << i++ << endl;       // Output: 2
   cout << i << endl;         // Output: 3
   cout << j-- << endl;       // Output: 8
   cout << --j << endl;       // Output: 6
   return 0;
}
```

The Unary Sign Operators

The sign operator + has no real use as it only returns the value of the operand as it is. On the other hand, the sign operator – returns the operand value after inverting its sign.

Example: int n = -5; cout << -n; //Output: 5

Increment and Decrement Operators

The increment operator is used to increase the value of the operand by 1. For this reason, it cannot be used with constants.

Increment operators can be used in the form of prefix notation, or postfix notation. Taking i as a variable, the postfix notation written as i++ and prefix notation which is ++i both perform the function of i=i+1. The end result has the value of I increased by one.

Despite performing the same operation, the postfix ++ and prefix ++ function in different ways. By observing the values in both expressions, the difference between the two can be made apparent. When ++i is used, it means the value of i is already incremented before being applied. Conversely, in i++, the original value that is i is retained.

++i, the value of i is first incremented, after which it is applied.

i++ the original value of i is applied and then incremented.

While it may not be as apparent in simple expressions, it makes a noticeable difference in complex expressions. Therefore, when dealing with complex expressions, the difference between the postfix and prefix expressions must be noted.

The decrement operator -- changes the value of the given variable or operand by reducing it by 1. The prefix and postfix notations can also be applied to this operator which function in the same way as the increment notations except it perform the operation i=i-1.

Precedence

When multiple operators are to be evaluated, their order is decided by the operator precedence, after which the operators and operand are grouped accordingly. If the table opposite is studied, it can be seen that the operator ++ has the highest precedence while “/” has higher precedence than “-.”

Example: (val++) – (7.0/2.0)

The result is 1.5, keeping in mind the fact that val is incremented later.

If any two or more operators have equal precedence, the evaluation of the expression is determined by column three of the table.

Example: 3 * 5 % 2   is equivalent to   (3 * 5) % 2

Assignments

The process in which we assign something to an expression or variable is known as assignment. Before we discuss the topic of assignment in detail, let’s first see how they are used in a practical C++ program.

```
// Demonstration of compound assignments
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   float x, y;
   cout << "\n Please enter a starting value: ";
   cin >> x;
   cout << "\n Please enter the increment value: ";
   cin >> y;
   x += y;
   cout << "\n And now multiplication! ";
   cout << "\n Please enter a factor: ";
   cin >> y;
   x *= y;
   cout << "\n Finally division.";
   cout << "\n Please supply a divisor: ";
   cin >> y;
   x /= y;
  cout << "\n And this is "
       << "your current lucky number: "
                              // without digits after
                              // the decimal point:
```

```
            << fixed << setprecision(0)
            << x << endl;
      return 0;
   }
```

Simple Assignments

When assigning a variable value to an expression, the simple assignments use the assignment operator written as =. The value is written on the right of the assignment operator "=" while the variable the value is assigned to is on the left side.

Example: z = 7.5;

```
y = z;
x = 2.0 + 4.2 * z;
```

As seen in the example, the variables are expressed as "x," "y" and "z" on the left; the values assigned to them are on the right. In order of evaluation, the assignment operator has low precedence. This can be seen in the last example where the expressions on the right side are evaluated first, and the result produced by the calculation is assigned to the variable on the left side.

The assignment operator can be considered as an expression in itself, and the value of this expression is the value assigned to the variable.

Example: sin (x = 2.5);

As shown in the example, the value of 2.5 is first assigned to the variable "x," which is then passed as an argument to the function.

If multiple assignment operators are used, it is possible to evaluate them from right to left.

Example: i = j = 9;

The evaluation of the expressions in this example starts from the right, i.e., the value 9 is first assigned to "j" and then to "i."

Compound Assignments

Another kind of operator besides the simple assignment is the compound assignment operator. It is used to perform the operations of arithmetic and assignment at the same time.

Example: i += 3; is equivalent to i = i + 3;

i *= j + 2; is equivalent to i = i * (j+2);

The precedence of compound assignment is similar to simple assignment operators and is set to be low. Hence the compound assignments are implicitly placed in parentheses which is demonstrated in the second example.

The binary arithmetic operators and bit operators are used for composing compound assignment operators such as +=, -=, *=, /=, and %=.

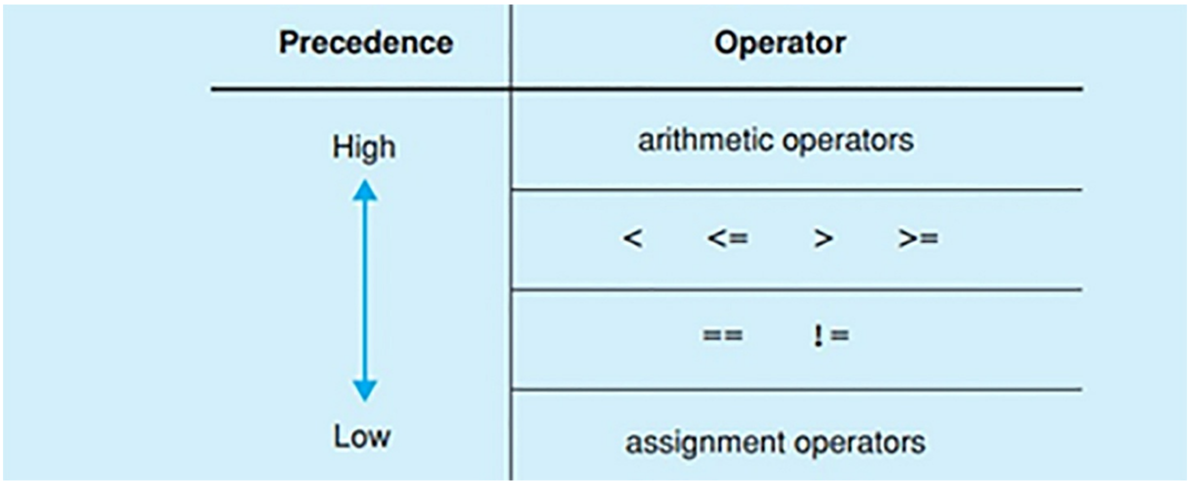
When evaluating a complex operation, the presence of assignment operators or increment(++)/decrement(--) operators can significantly modify a variable. This modification is referred to as side effect and can usually lead to errors. Therefore, the use of side effects should be avoided as much as possible so that the program readability is not impaired.

Relational Operators

Here's a table showing all of the relational operators available for use in C++.

Operator	**Significance**
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Unequal to

Like unary operators, relational operators also have different precedence among themselves. This has been highlighted in the table shown below:



Here's a brief example of the use of relational operators for comparison purposes:

Comparison	**Result**
4 >= 5	False
1.5 > 2.3	False
2.9 > 2.2	True
2 * 10 != 18	True

The Result of Comparison

The comparison operators are bool type expressions that are used to compare two values. The value may be "true," which means the comparison is correct, or it may be "false," meaning the comparison is incorrect.

Example: length == circuit //false or true

In the given example, if the number value of both length and circuit are the same, then the comparison is correct, and the value of this comparison expression is "true." On the other hand, if the length and circuit have different values, then the value assigned to this expression will be "false."

When comparing two individual characters, the character set must be carefully selected because the characters are compared based on their character codes.

Example: ‘A’ < ‘a’ //true, since 65 < 97

In the example, the ASCII code is used, and hence, the value turns out to be true.

Precedence of Relational Operators

In order of evaluation, the precedence for relational operators is approximately in the middle. It is lower than the arithmetic operators but higher than the assignment operators.

Example: bool flag = index < max – 1;

As seen in the example, the arithmetic expression max -1 is evaluated first. The result is then compared with the index, and finally, the value is assigned to the flag variable.

Example: int result;

result = length + 1 == limit;

In this case, the operation length + 1 is performed before the other expressions and is then compared with the limit variable. The result of this relational operation is then assigned to the result variable. Due to the result type being int type, the value of the end result is numerical, not true or false. So if the result is false, it is given a value of 0, and if it is true, then the value is 1.

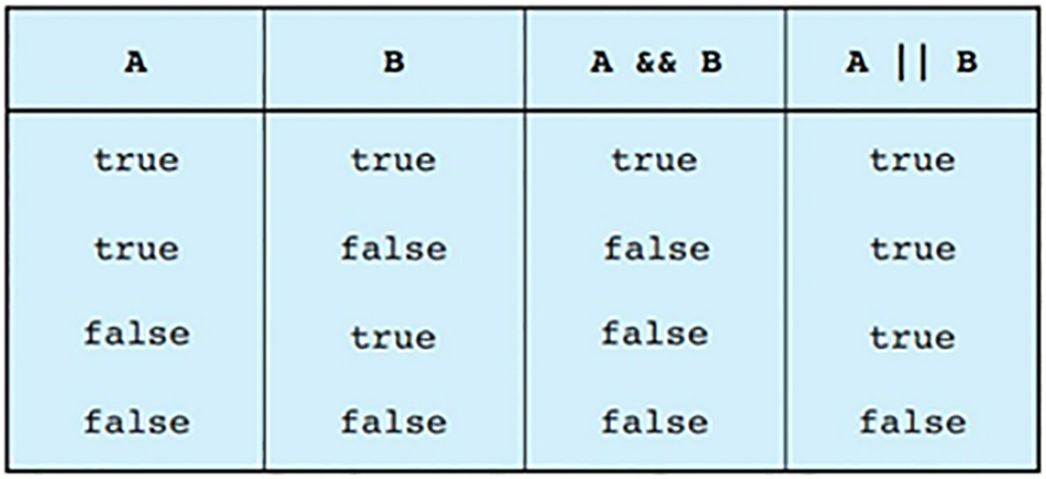
It is quite common to perform operations where the assignment operator is given precedence, and the other operations follow after. It is achieved by enclosing the assignment expression in parentheses.

Example: (result = length + 1) == limit

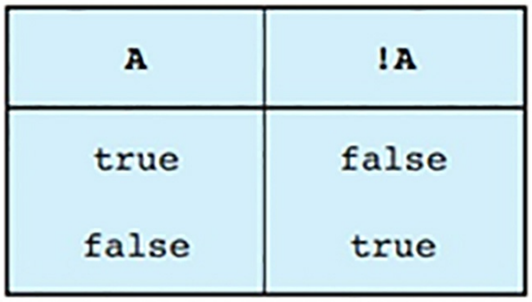
In this case, the value of length + 1 is first assigned to the result variable and is compared with the limit afterward.

Logical Operators

Before we go into a detailed discussion regarding logical operators, here’s a truth table for the logical operators alongside examples showcasing logical expressions. This is necessary to build background knowledge.

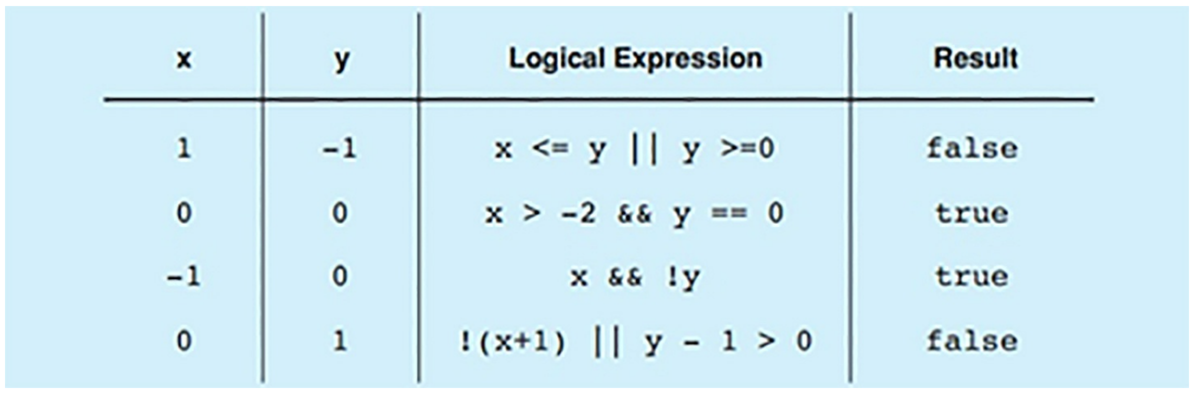


A	B	A && B	A || B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false



A	!A
true	false
false	true

(Truth Table)



x	y	Logical Expression	Result
1	-1	x <= y || y >=0	false
0	0	x > -2 && y == 0	true
-1	0	x && !y	true
0	1	!(x+1) || y - 1 > 0	false

The Boolean operators AND &&, OR || and NOT ! are all a part of the logical operators. The main use of these operators is to make compound conditions as well as carry out conditional execution of a program while relying on the multiple conditions set by the user. Similar to relational expressions, logical expressions are checked to see if they are correct or incorrect, and then a value between true or false is assigned to the end result.

Operands and Order of Evaluation

The Boolean type operators usually make use of bool type operands. However, arithmetic type operands, as well as any operands that are convertible to bool type, are also used. As such, if the operand used has a value of 0, it is converted to false. If its value is anything other than 0, then it is taken as true.

The OR operator || yields the relational result true if at least one of the operands is true. If both are false, then the result will be false as well.

Example: (length < 0.2) || (length > 9.8)

The value of OR expression given above will be true if the value of the length is less than 0.2 or greater than 9.8.

The value in an AND operator && expression will be returned as true only if both the operands are true and otherwise the result will be false.

Example: (index < max) && (cin >> number)

If, in the above expression, the index is proven to be less than max, and the number is input successfully, then the result produced by the operator will be true. On the other hand, if the index is not less than max, the number will not be read or input by the program. The logical operators AND '&&' as well as OR '||' have an order of evaluation that is fixed. C++ starts the evaluation from left to right and ignores the unnecessary operands, such as when the result is established while evaluating the left operand, the right operands are not evaluated.

The logical operator ! is the NOT operator that works with only one operand located on its right. It checks the value of the operands, and if it is true, then it will be inverted by the NOT operator, and the Boolean value will be returned as false. Similarly, if the value of the variable flag is false or 0, then it will be returned as true. Therefore, NOT operator works by returning the opposite value of the variable flag it evaluates.

Precedence of Boolean Operators

When considering the order of evaluation, the AND operator && is set to have higher precedence than the OR operator ||. Both these logical operators have higher precedence than the assignment operators and lower than other previous operators.

The NOT operator ‘!’, being the unary operator, among others, has higher precedence in the evaluation order.

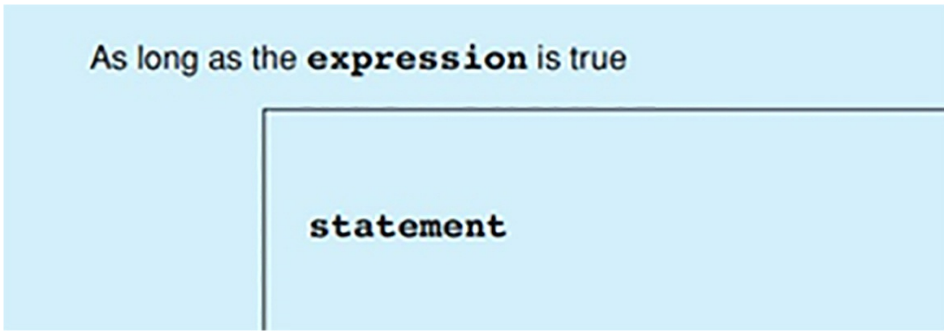
Chapter 5

Controlling the Flow of a Program

In a program, we need to control the flow of data and information. A program that has a data flow in one continuous direction, i.e., the flow of data is in the direction of the program's structure, then the resulting program is limited and one-dimensional. To increase the functionality and capability of a program, we need to control its flow effectively. To do this, we need to use certain statements such as loops (while, for, do-while), using conditional operators (if, then, if-else, etc.) and jumps (goto, continue and break).

The 'While' Statement

Here's a structural diagram of the **while** statement.



Similarly, let's first observe a program demonstrating the use of the **while** statement for controlling its flow and then proceed to break it down and understand it.

```
// average.cpp
// Computing the average of numbers
#include <iostream>
using namespace std;
```

```
int main()
{
   int x, count = 0;
   float sum = 0.0;
   cout << "Please enter some integers:\n"
            "(Break with any letter)"
        << endl;
   while( cin >> x )
   {
     sum += x;
     ++count;
   }
   cout << "The average of the numbers: "
        << sum / count << endl;
   return 0;
}
```

The iteration statements or loops repeat a set of instructions that are supposed to repeat for a certain number of times. The statements or set of instructions that are to be repeated are referred to as loops bodies. The three language elements, namely while, do-while and for are the ones responsible for expressing iteration statements. The controlling expression is a condition that limits the number of times a loop is repeated, i.e., until the expression is no longer true. The while and for statements verify whether the expression is true or not before the loop body is executed. In contrast, the do-while statement evaluates the controlling expression after the statement is executed once.

The syntax of while statement is expressed as:

Syntax: while (expression)

statement // loop body

The controlling expression is evaluated before C++ enters a loop made by any of the iteration statements. If the expression is verified to be true, the loop body is executed once. After, the controlling expression is evaluated again. The process is repeated if the expression is true, but if it evaluates to false, the program exits the loop and executes the statement coming after the while statement.

The program readability can be improved by starting the coding of the loop body from a new line in the source code and adding an indent to the statement.

Example: int count = 0;

```
while ( count < 10)
    cout << ++count << endl;
```

The given example suggests that usually, Boolean expressions are used for controlling expressions, but it is not strictly limited to only Boolean expressions. The expressions that are convertible to bool type, as well as arithmetic expressions, can also be used as controlling expressions in iteration statements. If the value in these expressions is 0, it is interpreted as false, while any other value is converted to true.

Building Blocks

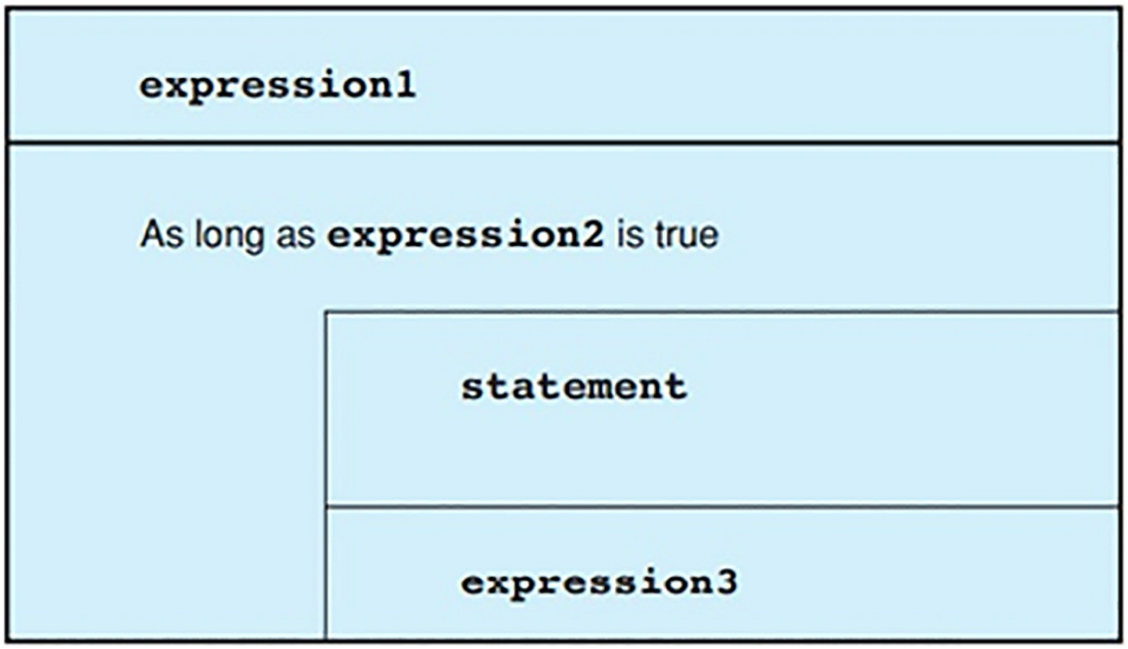
When more than one statement needs to be repeated in a loop body, each statement must be placed in parentheses. These are referred to as a block. Whenever a statement is required in syntax, a block can be used in the expression as a block is syntactically equal to a statement.

In the following sample program, the average of a sequence of integers is calculated. The loop body contains two statements to be repeated in the process. Hence, both of these statements are placed in a block marked by parentheses.

The controlling expression in the program is ‘cin >> x’ and holds to be true when the input is an integer. During the conversion of ‘cin >> x’ to bool type, if the input is valid, then the result would be true, and if it is invalid, the result is returned as false, and the loop is terminated. In the given case, a valid input would be any integer. In contrast, an invalid integer could be a letter that makes C++ exit the loop and execute the statement following after the loop body.

The ‘For’ Statement

Just like in the previous section, let’s first look and understand the structural diagram of the '**for** statement' in a program and see how it is actually implemented in a program as well. The structure of the **for** statement is as following:



Similarly, the implementation of the '**for** statement' in a program has also been demonstrated below:

```
// Euro1.cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   double rate = 1.15;      // Exchange rate:
                           // one Euro to one Dollar
   cout << fixed << setprecision(2);
   cout << "\tEuro \tDollar\n";
   for( int euro = 1; euro <= 5; ++euro)
     cout << "\t " << euro
          << "\t " << euro*rate << endl;
return 0;
}
```

The output of this program is as follows:

```
Euro     Dollar
1        0.95
2        1.90
3        2.85
4        3.80
5        4.75
```

Initializing and Reinitializing

Example:
```
int count = 1;       // Initialization
while( count <= 10)  // Controlling
{                    // expression
 cout << count
    << ". loop" << endl;
++count;             // Reinitialization
}
```

The expressions or elements that control the repetition of the statements in the loop are typically placed in the loop header. You saw this in the above example, which can be considered as a ‘for statement.’

Example:
```
int count;
for ( count = 1; count <= 10; ++count)
    cout << count
        << ". loop" << endl
```

Any expression can control the initialization and reinitialization of a loop in a for statement. Hence, the for statement has the following syntax:

Syntax:
```
for ( expression1; expression2; expression3 )

 Statement
```

As seen in the syntax, expression 1 is executed only once at the beginning of the loop so it can be initialized. The next expression 2 is evaluated before the loop body is implemented and is the controlling expression of this statement.

- In case the value of expression 2 is returned as false or 0, the loop is terminated.

- In case the value of expression 2 is true, the execution of the loop body takes place.

The loop is repeated when expression 3 reinitializes the statement and returns to expression 2 to verify the value again.

The loop counter placed in expression 1 can only be used within the loop, and not after the loop is terminated.

Example: for (int i = 0; i < 10; cout << i++)

```
;
```

The example given suggests that some loops may even have empty statements because all of the required statements are located in the loop header. Though this method works, the readability of such a loop body is significantly worse than the one that has a line of its own in the empty statement.

```
// EuroDoll.cpp
// Outputs a table of exchange: Euro and US-$
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   long euro, maxEuro;          // Amount in Euros
   double rate;                 // Exchange rate Euro <-> $
   cout << "\n* * * TABLE OF EXCHANGE "
        << " Euro – US-$ * * *\n\n";
   cout << "\nPlease give the rate of exchange: "
            " one Euro in US-$: ";
   cin >> rate;
   cout << "\nPlease enter the maximum euro: ";
   cin >> maxEuro;
          // --- Outputs the table ---
                                        // Titles of columns:
```

```
    cout << '\n'
         << setw(12) << "Euro" << setw(20) << "US-$"
         << "\t\tRate: " << rate << endl;
                                                  // Formatting US-$:
    cout << fixed << setprecision(2) << endl;
    long lower, upper,                  // Lower and upper limit
          step;                         // Step width
         // The outer loop determines the actual
         // lower limit and the step width:
    for( lower=1, step=1; lower <= maxEuro;
                          step*= 10, lower = 2*step)
         // The inner loop outputs a "block":
     for( euro = lower, upper = step*10;
          euro <= upper && euro <= maxEuro; euro+=step)
       cout << setw(12) << euro
            << setw(20) << euro*rate << endl;
    return 0;
}
```

In a ‘for statement,’ any from the expressions 1, 2, and 3 can easily be omitted. The exception to this is that there must at least be two semicolons in the loop. Thus, following this proclamation, the shortest form of a for statement would be written as:

Example: for (; ;)

In this example, the controlling expression is the condition that expression 2 must be missing. Therefore, in this case, the value is always returned as true, and the loop becomes infinite.

Example: for (; expression ;)

This expression of a for statement behaves in the same way as a while statement; the loop continues as long as the controlling expression is verified to be true.

The Comma Operator

The comma operator is used to separate two expressions, which are included in a code where only one statement is expected. An example would be the several variables set as initializers in a looping header of a ‘for statement.’ The comma operator has the following syntax:

Syntax: expression1, expression2 [, expression3 ...]

The set of expressions is evaluated from left to right, and when they have to be evaluated for a single value, only the right-most expression is considered.

Example: int x, i, limit;

```
For ( i=0, limit=8; i < limit; i += 2)
   x = i * i, cout << setw (10) << x;
```

The comma operator separates the different calculations in the above example. The assignments for i and limit are comma-separated, followed by the calculation and output of the value of x all in a single statement.

The precedence of the comma operator in the order is the lowest, placing even lower than the assignment operator. Hence, the use of parentheses is not needed, as in the example above.

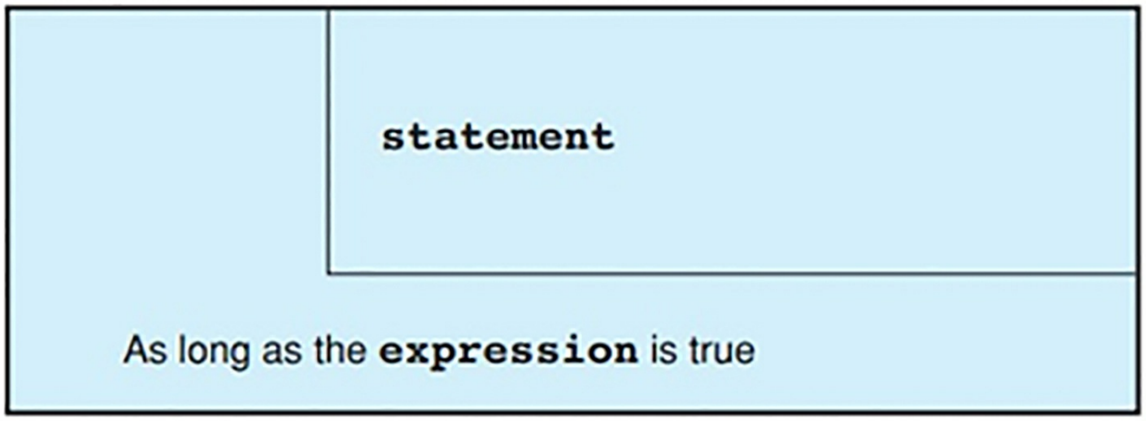
The last expression in a statement containing commas determines the type and value of a comma operator.

Example: x = (a = 3, b = 5, a * b);

The expressions starting from the left are evaluated first, and then the value of a * b is assigned to x.

The ‘do-while’ Statement

The structure of the do-while statement is as follows:



The practical implementation of the do-while statement in a program to control its flow has been demonstrated below:

```
// tone.cpp
#include <iostream>
using namespace std;
const long delay = 10000000L;
int main()
{
   int tic;
   cout << "\nHow often should the tone be output? ";
   cin >> tic;
   do
   {
     for( long i = 0; i < delay; ++i )
         ;
     cout << "Now the tone!\a" << endl;
   }
   while( --tic > 0 );
   cout << "End of the acoustic interlude!\n";
return 0;
}
```

In the do-while statement, the loop is always executed at least once, and the controlling expression is evaluated after the loop. Therefore, this iteration statement is controlled by its footer as opposed to the other two statements, which are controlled by their headers.

Syntax: do

```
statement
while ( expression);
```

The do-while evaluates the controlling expression after the loop body has been executed once, and if the value of the expression is returned as true, the loop is repeated again; meanwhile, the false result will terminate the loop.

Nesting Loop

Nesting loops mean that a different loop is nested inside the loop body, i.e., a loop inside a loop. At most, 256 levels of nesting are allowed in C++ according to the ANSI standard.

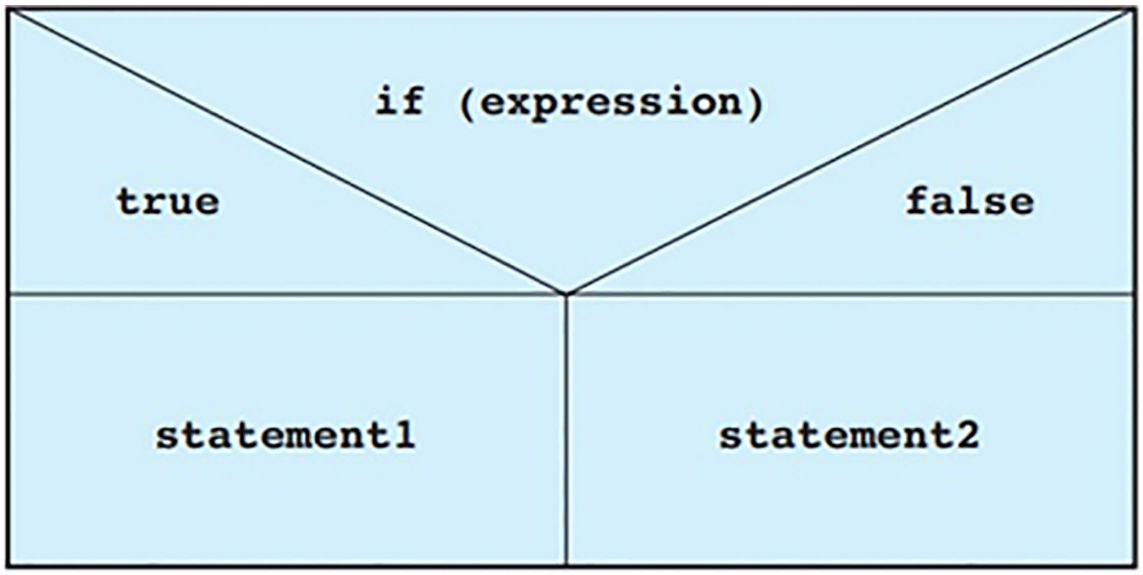
The sample program, as shown, gives an output of several tones as determined by the user input.

There are two loops in the given program where one loop is nested in another. Whenever the outer loop (do-while statement) is repeated, a short break occurs during the process during which the inner loop (for statement) is executed. In the inner loop, the value of i is incremented from 0 to delay value.

The output of this program is text and a tone, which is generated by outputting the control character BELL (ASCII code 7). This control character is characterized as the escape sequence /a. The do-while statement used in the program outputs the tone even if the input is a 0 or a negative number.

Selections of 'If-Else' Statements

The structural diagram of the **if-else** statement is as follows:



Similarly, the implementation of the **if-else** statement has been demonstrated in the program shown below:

```
// if_else.cpp
// Demonstrates the use of if-else statements
#include <iostream>
using namespace std;
int main()
{
   float x, y, min;
   cout << "Enter two different numbers:\n";
   if( cin >> x && cin >> y)       // If both inputs are
   {                          // valid, compute
      if( x < y ) // the lesser.
           min = x;
     else
           min = y;
     cout << "\nThe smaller number is: " << min << endl;
   }
   else
     cout << "\nInvalid Input!" << endl;
return 0;
}
```

The if-else statement is used when a choice is to be made between two statements, based on the conditions they fulfill.

Syntax:
```
if ( expression )
    statement1
[ else
    statement2 ]
```

The expression is evaluated first to verify whether the condition is fulfilled. The result is returned as true or false. If it is true, then the statement1 is executed, and statement2 is processed in other cases only if an else branch exists. In case the result is false, then statement1 is ignored, and statement2 is executed. However, if there is no else statement or it is also false, then the control skips to the statement following after the if-else statement.

Nested if-else statements

Considering the situation where more than one if-else statements are used in a program, multiple if-else statements can be nested in each other. Some ‘if statements’ do not have an ‘else’ branch associated with them. So, the ‘else’ branches are set to be associated with the nearest preceding ‘if statement’ that does not have an else branch.

Example:
```
if ( n > 0 )
    if ( n%2 == 1 )
        cout << " Positive odd number ";
    else
        cout << "Positive even number";
```

As visible from the example, the else branch is associated with the second if statement, which is indented. In case the else branch needs to be redefined and associated with another if statement, a code block is used.

Example:
```
if ( n > 0 )
  { if ( n%2 == 1 )
      cout << " Positive odd number \n";
    }
    else
      cout << " Negative number or zero\n";
```

Defining variables in if statements

A variable can be defined within the if statement and used for initialization. If the variable is converted to a bool type and returns the value true, then the expression in if statement will also be true.

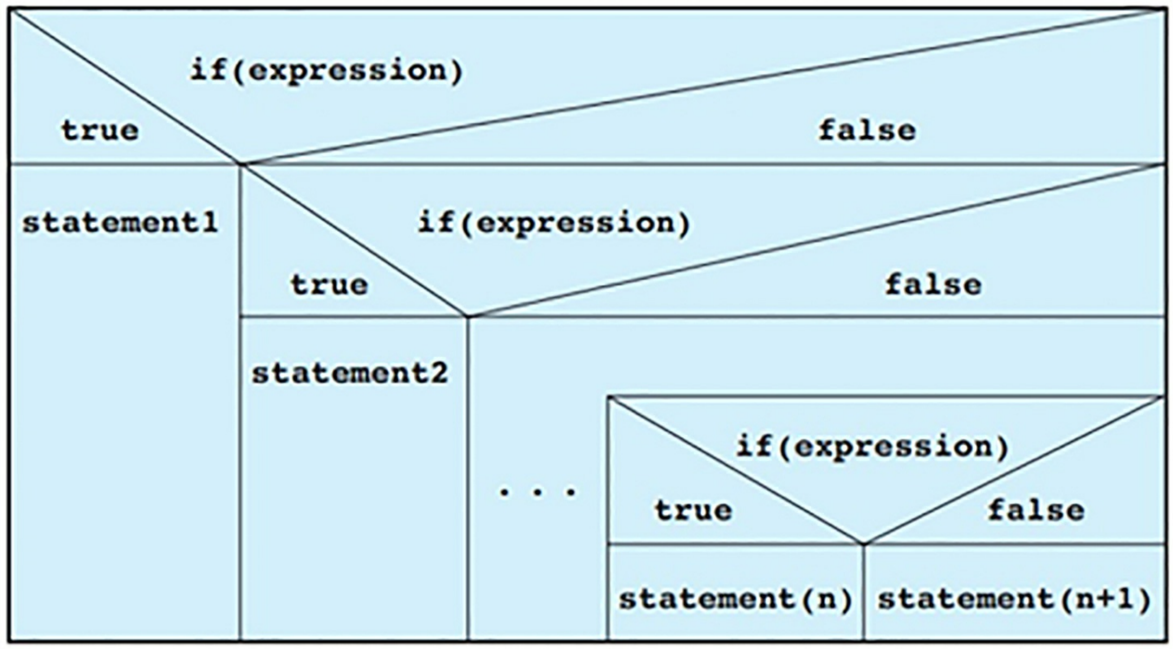
Example: if (int x = func ( ))

{ . . . } // Here to work with x.

The variable x in the example is initialized by the result of the function func ( ). For any value other than 0, the statement in the block is evaluated and executed. Once the program leaves the if statement, the variable x is no longer used.

Else-If Chains

The structural implementation of the **else-if** statement is as follows:



The implementation of the **else-if** statement has been demonstrated in the sample program shown below:

```
// speed.cpp
// Output the fine for driving too fast.
#include <iostream>
```

```
using namespace std;
int main()
{
   float limit, speed, toofast;
   cout << "\nSpeed limit: ";
   cin >> limit;
   cout << "\nSpeed: ";
   cin >> speed;
   if( (toofast = speed – limit ) < 10)
     cout << "You were lucky!" << endl;
   else if( toofast < 20)
     cout << "Fine payable: 40,-. Dollars" << endl;
   else if( toofast < 30)
     cout << "Fine payable: 80,-. Dollars" << endl;
   else
     cout << "Hand over your driver's license!" << endl;
   return 0;
}
```

Layout and Program Flow

If a program has multiple options to choose from, they can be executed selectively with the help of else-if chains, which are a series of if-else statements embedded within. The layout of these chains are:

```
if ( expression1 )
   statement1
else if ( expression2 )
   statement2
.
.
.
else if ( expression(n) )
     statement(n)
[ else statement (n+1)]
```

During the execution of the else-if chain, the expressions starting from the first one (expression1) are evaluated one by one as specified in the order. The first expression is verified, whether if it is true or false. If it is true, then the statement following it is performed, and the chain is terminated, but if it is false, the next expression is verified and so on.

If none of the expressions return the value true, then the else branch associated with the last ‘if statement’ is executed. In case this branch is left out, the program exits the else-if chain, and the statement coming afterward is evaluated.

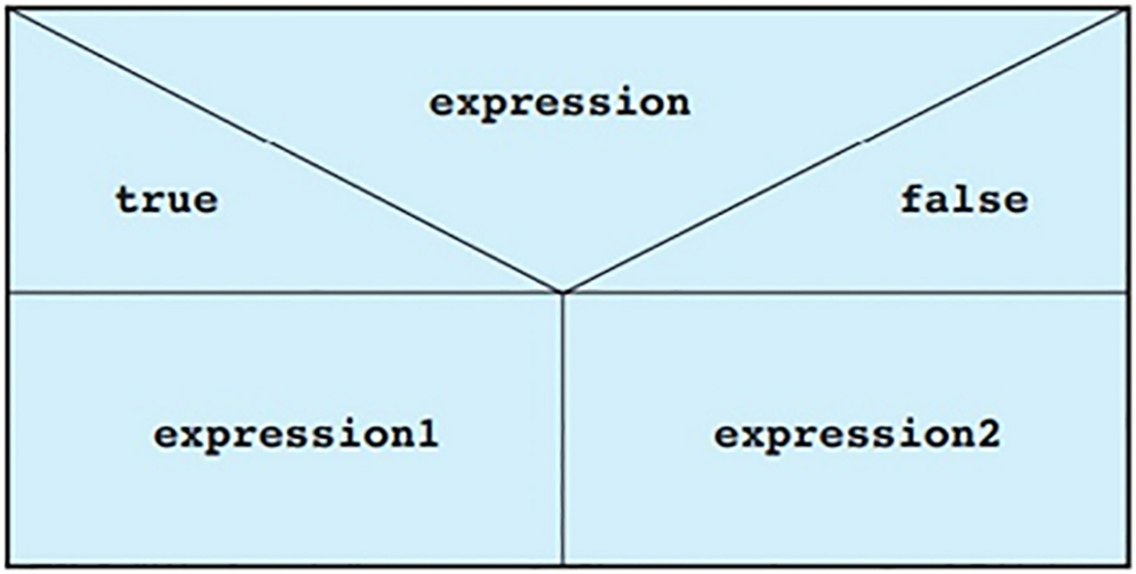
The Sample Program

It can be seen in the following sample program that an else-if chain is used in the coding to calculate the penalty of driving the vehicle over the speed limit. The result of this calculation is the penalty fine, which is displayed as the output on the screen.

The vehicle’s speed limit and the actual speed are input via the keyboard into the code. If the actual speed exceeds the speed limit, such as a car was seen to have the actual speed of 97.5 while the speed limit is 60, the first two expressions in the else-if chain are returned as false, and the last else branch is executed. This statement displays the message “Hand over your driver’s license” as the output on the screen

The Conditional Operators

The structural implementation of a conditional expression in a program is as follows:



A sample program demonstrating the implementation of conditional operators and expressions has been shown below:

```
// greater.cpp
#include <iostream>
using namespace std;
int main()
{
   float x, y;
   cout << "Type two different numbers:\n";
   if( !(cin >> x && cin >> y) )        // If the input was
   {                              // invalid.
     cout << "\nInvalid input!" << endl;
   }
     else
  {
     cout << "\nThe greater value is: "
          << (x > y ? x : y) << endl;
   }
   return 0;
}
```

The output of this program is:

```
Type two different numbers:
        173.2
        216.7
        The greater value is: 216.7
```

Conditional operators (?:) can be used as a concise alternative to the if-else statements. This selection mechanism is based on the fact that one of the two given values in a statement is selected as the output depending on the value of the associated condition.

The value selected and produced in this kind of expression depends on whether the value of condition is returned as true or false, and so, it is referred to as conditional expression.

Syntax: expression ? expression1 : expression2

The value of the conditional expression will either be expression1 or expression2. The expression or given condition is evaluated first. If the result value is returned as true, then expression1 is selected and evaluated, but if the result is false, then expression2 is chosen for evaluation.

Example: z = (a >= 0) ? a : -a;

The variable z in the given expression is to be assigned an absolute value of a. The first condition is that if the value of a is positive such as 12, then the variable z is assigned the number 12. But if it is a negative number like -8, the value of z would be 8.

The value obtained from the conditional expression is stored in the variable z, and hence, it can be considered to the if-else statement shown below:

```
if ( a > 0 )
    z = a;
else
    z = -a;
```

Precedence

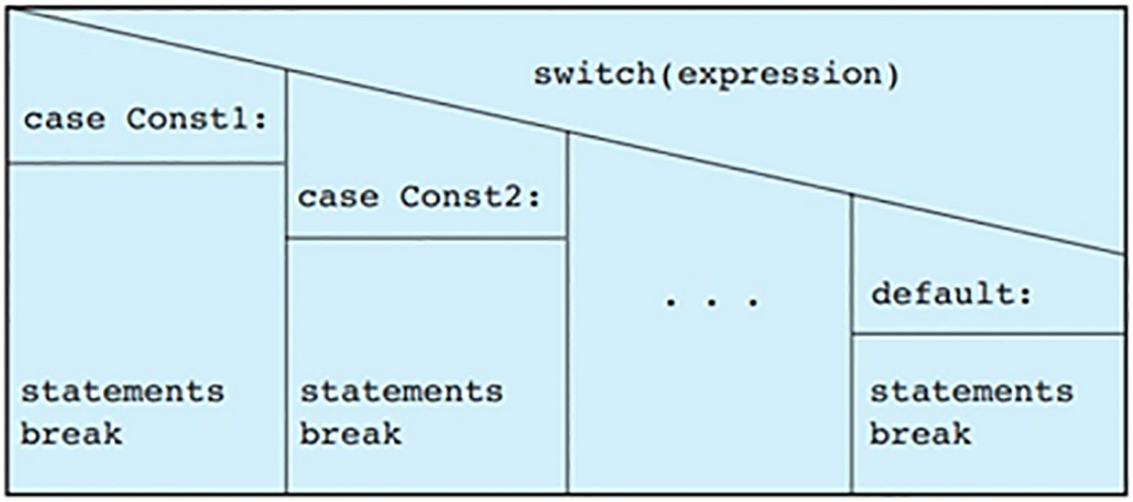
The only operator that uses three operands in the C++ language is the conditional operator. The brackets visible in the first example can be removed because the precedence of a conditional operator is higher than assignment operators and comma operators and lower than the rest.

The result produced by the evaluation of a conditional statement can be used directly without having to assign it. This can be understood by the following example, which has the condition that x should be greater than y. If x is a greater value, then the value of x is displayed as the output. Otherwise, the value of y will be printed.

For complex expressions, it would be better to use a variable to which the value of the conditional expression will be assigned so that the readability of the program can be increased.

The ‘Switch’ Statements

The structural implementation of a switch statement in a program is as follows:



A sample program demonstrating the implementation of switch statements and expressions has been shown below:

```
// Evaluates given input.
int command = menu();           // The function menu() reads
                                // a command.
switch( command )               // Evaluate command.
{
   case 'a':
   case 'A':
            action1();          // Carry out 1st action.
            break;
   case 'b':
   case 'B':
             action2();         // Carry out 2nd action.
             break;
   default:
            cout << '\a' << flush;  // Beep on
}                               // invalid input
```

Just as the else-if chain evaluates multiple statements in sequence, the switch statement also chooses between numerous alternatives. The difference is that it compares the expression given in the beginning with all the other constants or statements.

```
switch ( expression )
```

```
{
      case const1: [ statement ]
                  [ break; ]
      case const2: [ statement ]
                  [ break; ]
      .
      .
      .
     [ default : statement ]
}
```

Before implementation of the switch statement, it is mandatory to consider whether the expression that needs to be evaluated and the constants to which it will be compared to are all of the integral type (such as Boolean values or character constants). Then the result of the expression evaluation is compared to const1, const2, and so on, written in the case labels. Each of these constants must be different.

When the result value finds a match among the multiple case constants, the program moves on to the selected case label and continues onwards. After this step, the case labels are not required anymore.

The unnecessary execution of case labels after the switch statement is implemented is prevented by putting the break statement after each constant statement. This makes the program leave the statement unconditionally.

In case the expression result value does not match with any of the constants, the program branches to the default label, which may not necessarily be the last label. If additional case labels need to be added to the switch statement, they can be placed after the default label. If the default label is not defined, then nothing happens, and the program moves on to execute the next operator.

Differences between the switch and else-if chains

In terms of versatility and usefulness, else-if chains are better than switch statements because every kind of selection can be programmed using the else-if chain. The disadvantage to the else-if chain is that the integral values of the expressions need to be constantly compared to several possible values. So, for cases like this, it is better to use a switch statement.

Comparing the readability of the else-if chain and the switch statement is the given example, it is clear that switch statements are easier to read and hence should be used wherever it is possible.

Chapter 6

Arithmetic Data Type Conversions

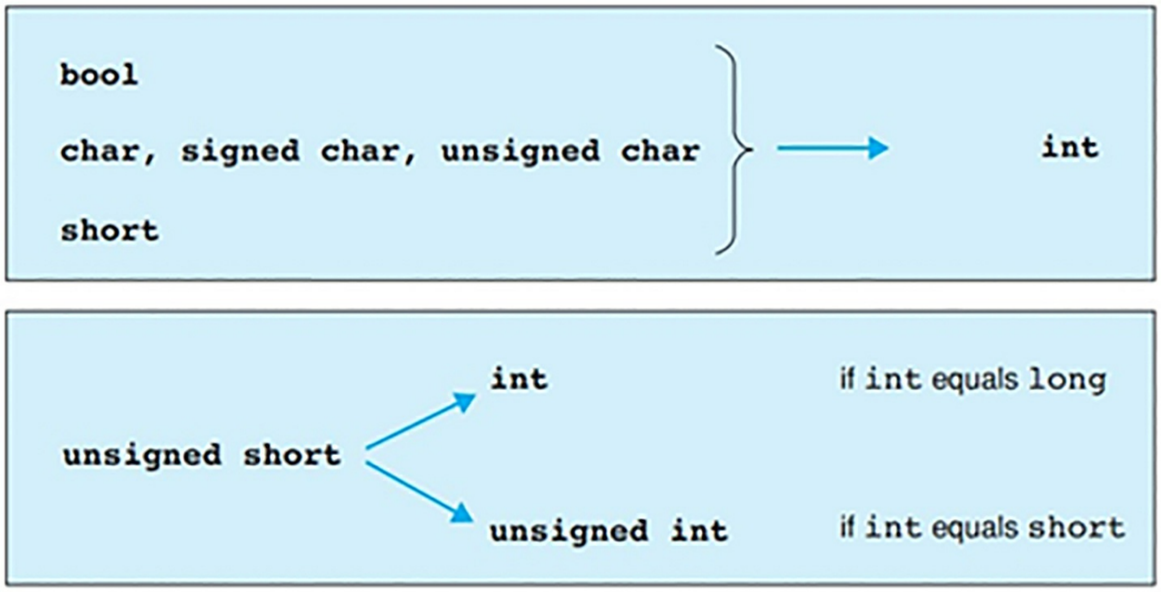
In programming, it is very common to have arithmetic types which are different from each other. Just as the types are different, it’s a given that we cannot perform mathematical operations on inputs belonging to two different arithmetic types. When dealing with such types, we need to convert one of them into the same type as the other. In this chapter, we will learn all about these conventions and rules to arithmetic types of conversions.

However, be mindful when going through this chapter, as you will often find yourself dealing with different data types (especially arithmetic). In such cases, it is important to know how to deal with them. Moreover, data type conversions are a very important and very fundamental skill when it comes to C++ programming. This is because big programming projects often involve huge amounts of data streams from different sources and are very intricately intertwined. This means that the inputs and outputs of such a complex program begin to form an ecosystem of their own as they are used and reused by different classes, objects, and functions. As such, it is not always the case that such data being circulated through the program is going to be the correct type. This is why implementing data type conversions is very important to allow the program to work efficiently.

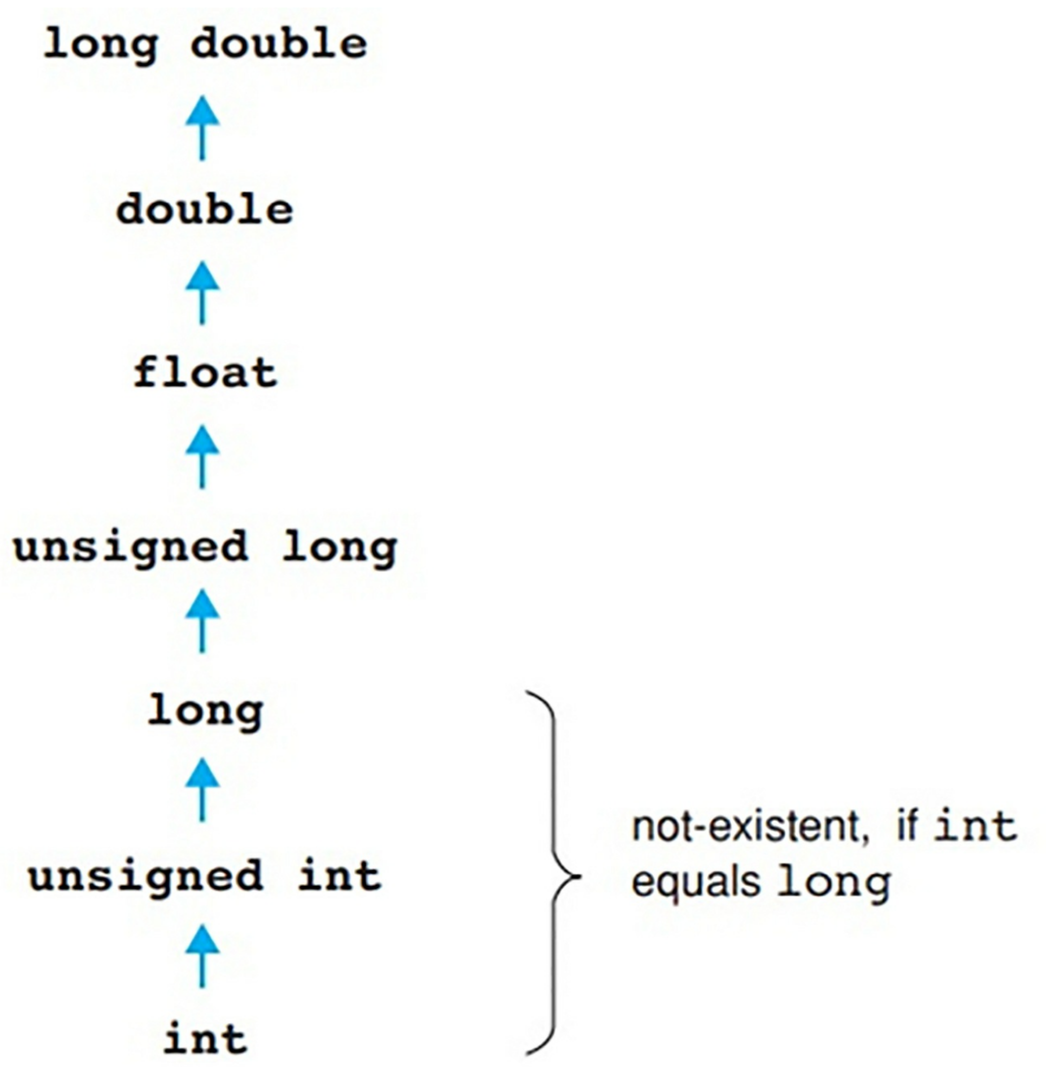
In this chapter, we will chiefly learn about **implicit type conversions** and briefly discuss some additional type conversions at the end of this chapter.

Implicit Type Conversions

Before we go into the main discussion, we need to set up the context. The following figures that have been shown below highlight the integer promotion and hierarchy of **int** type respectively:



(Integer Promotion)



(int type hierarchy)

In many of the C++ programs, we will occasionally see a single expression containing different arithmetic types. This mixed-up arithmetic types within the same expression mean that the operands of the corresponding operator belong to the different types that have been highlighted in the expression. The reason why this does not cause an error in the program while it tries to perform the corresponding operation is due to the compiler performing an implicit type conversion by itself.

In an implicit type conversion, the requirement of the operation which needs to be performed is first considered. This means that we first check the type needed for the operation to proceed. This type that is determined is a common ground between the two different types between which we are performing the implicit type conversion. Once a common type through which we can perform the required operation is decided, the values for both operands are assigned this type.

A general rule in implicit type conversion is that a ‘**smaller**’ type is the one which is subject to conversion. The smaller type is generally converted to the **larger** type of the two operands. However, there is an operator that is exempt from this rule, and this operator is the **assignment operator**. We will discuss the assignment operator in the upcoming sections of this chapter in detail.

If an arithmetic operation is performed, then the result obtained will be in the same type as the one which was specified for the operation to be performed. Note, however, that no arithmetic operation can be performed on one value, there at least needs to be two values. On the other hand, regardless of the type of operands used, a comparison expression will always be a **bool** type.

Now let’s talk about the two figures shown at the beginning of this section.

Integer Promotion

In this type of conversion, the main focus is to conserve the value of the original type when it is converted into the **int** type. For instance, a **bool** type containing a value which is either a **True** or **False** value, when converted through **integer promotion**, will have its values changed to “1 for True” and “0 for False”. In this way, the original value is preserved coming into the **int** type.

Moreover, the integer promotion type conversion is performed on expressions that are:

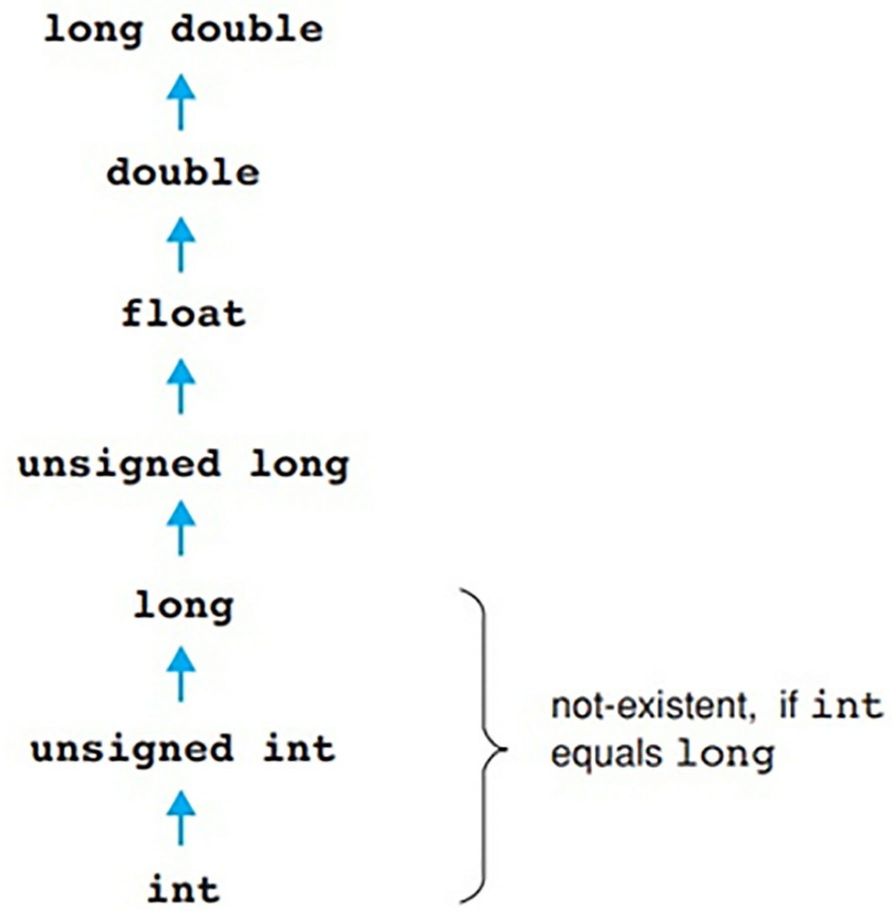
- **bool, short, signed char** and **unsigned char**. Expressions containing these types are converted to the **int** type.
- If an expression is an **unsigned short** type, then it is converted to the **int** type only if it is bigger than the **unsigned short**. If the **int** is not greater than the **unsigned short** type, then it will be converted to **unsigned int** as well as in other cases.

In short, C++ will always prioritize the **int** type values in operations that involve calculations. For example, let’s say that we are comparing a **char** variable **f** and the value **‘b,’** these values will be first converted to the **int** type before performing a calculation such as:

```
c < 'a'
```

Performing Some of the Usual Arithmetic Type Conversions

In some cases where we end up with operands that differ in arithmetic types even after performing integer promotion type conversion, we will need to perform an implicit type conversion using **int type hierarchy**. The diagram highlighting this implicit type conversion has already been shown previously, here’s a reminder of what hierarchy type conversion refers to.



When further performing an implicit type conversion using type hierarchy after integer promotion, we mainly convert the two operands into a type that holds the highest position in the hierarchy. When these two implicit type conversions (integer promotion and type hierarchy) are performed, they are collectively known as *“Usual Arithmetic Type Conversions.”* For example, carefully assess the type conversion shown below:

```
short size(512);  double res, x = 1.5;
res = size / 10 * x;       // short -> int -> double
```

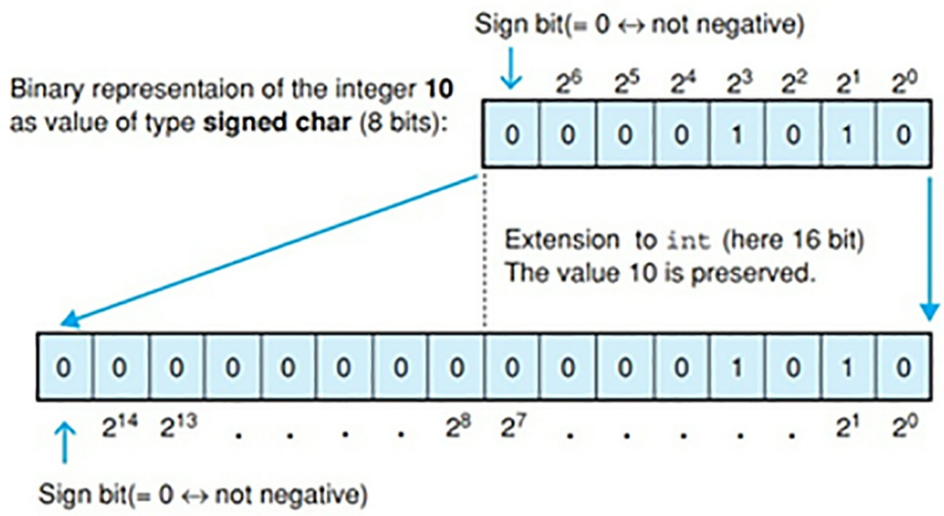
In this example, we are performing two mathematical operations, i.e., division, and multiplication. The original type of the **size** variable is **short**. Before we can perform the division operation in **size/10**, the type of the **size** variable is promoted from **short** to **int**. Once we obtain the result of this operation (which would be 50), the type of this result is further converted to the **double** type by implicit type conversion of hierarchy. Once the type of the result is converted to **double**, only then can we perform the multiplication operation with **x.** This is because the value of x is of a **double** type, and the resulting value of the integer division is of the **int** type.

Generally, 'Usual Arithmetic Type Conversions' are most often sought to be performed on conditional operators such as (?:) and all of the binary operations. However, the only condition specified for the Usual Arithmetic type conversions is that the corresponding operands should be of the Arithmetic type.

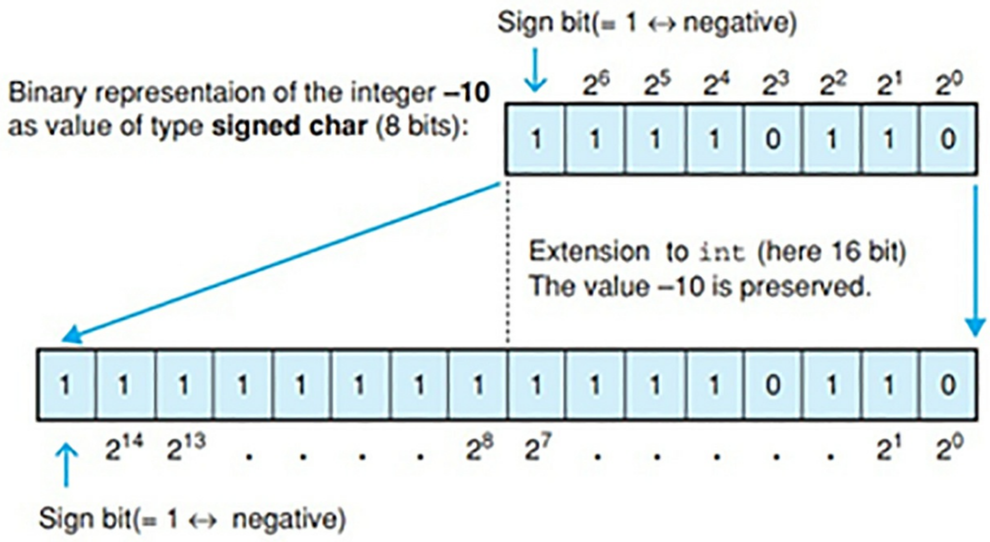
Now let's discuss a few cases where we can apply the Usual Arithmetic Type Conversions.

1. Converting Signed Integers

As we know that an integer can refer to either a positive number or a negative number, hence we will need to address both states of signed integer conversions. The figure shown below accurately demonstrates and elaborates on the conversion of a positive number.


Sign bit(= 0 ↔ not negative)
Binary representaion of the integer **10** as value of type **signed char** (8 bits):
2^6 2^5 2^4 2^3 2^2 2^1 2^0
0 0 0 0 1 0 1 0
Extension to int (here 16 bit)
The value 10 is preserved.
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
2^{14} 2^{13} . . . . 2^8 2^7 . . . . . . 2^1 2^0
Sign bit(= 0 ↔ not negative)

For the conversion of negative numbers (integers), for instance, -10, we first need to calculate this binary number's bit pattern and then generate a corresponding binary complement. This process has been demonstrated below:



It is important to remember that the interpretation of the value of a negative number is subject to change if the type is **unsigned**. In the following sections, we will discuss the varying procedures for type conversions relative to specific types.

2. Conversion of an Unsigned Type to a Larger Integral Type

For converting an unsigned type to a larger integral type, we will need to perform a process known as **‘Zero extension**.’ In zero extension, we take the number that we want to convert and calculate its bit pattern. Once the bit pattern has been calculated, we simply expand the bit pattern to make it match the type in which it is being converted. This is done by simply adding zeroes to the bit pattern from the left-hand side. In other words, the zero extension process involves the expansion of a number’s bit pattern by adding zeroes to it, to make its size match the destination type. For example,

unsigned char to int or unsigned int

3. Conversion of a Signed Type to a Larger Integral Type

First, we will talk about the case where the new type is also **signed**. To represent signed integers, we need to generate a binary complement of the corresponding numbers. To preserve the original value of the numbers in such cases, we need to perform a process known as ‘**Sign extension.’** In the sign extension process, we expand the integer’s original bit pattern by padding the sign bit from the left direction. For example

char to int, short to long

In this way, the bit pattern is expanded to match the length of the destination type. This process has also been depicted in the figures shown previously.

Now let’s talk about the scenario where the new type is **unsigned**. When dealing with such integers, the original value of the negative number cannot be preserved or retained. However, if the type in which we want to convert the number has a bit pattern that is of the same length, then the bit pattern is retained as there is no need to perform bit pattern extensions. But even if the bit pattern is retained, this does not mean that it will be interpreted the same way. In such cases, the sign bit will become insignificant, i.e., it will lose its significance. So, if the type convert destination is longer than the original type, then a sign extension is performed to generate a new bit pattern, and this bit pattern is then interpreted as **unsigned**. For example

char to unsigned int, long to unsigned long

4. Conversion of an Integral Type to a Floating-Point Type

In this type of conversion, the original value of the number is preserved while being converted into a floating-point number featuring an exponent. However, there are cases where the original number may be rounded-off during conversion. This is evident when converting a value of the **long** or **unsigned long** type to a **float** type value. For example.

int to double, unsigned long to float

5. **Conversion of a Floating-Point Type to a Bigger Floating-Point Type**

The core process remains the same; we are simply converting the **float** type to a **double** type or the **double** type to an even larger type, such as the **long double** type, as shown below.

float to double, double to long double

Throughout this type of conversion, the original value of the number is retained.

Implicit Type Conversions with Assignment Operators

Even if we are not familiar with assignment operators, everybody has come across these operators when exploring programming. We have even used assignment operators in the C++ programs shown in this book. An assignment operator allows you to assign a value to a variable. There are several shorthands and operands of this tool in programming, but the most common one is the equal sign (=).

In this section, we will be discussing how implicit type conversions can be performed during assignments. You might find it interesting that during assignments, arithmetic types can also be brought into the process. In a sense, the compiler acts as a mediator balancing out the left and right-hand sides of the assignment operands. In other words, the value's type on the right-hand side of the assignment operator is adjusted to match the value's type on the left-hand side of the assignment operator.

However, not all assignments are as simple as x = 2. Assignments can also have multiple statements (including many calculation operations, etc.) and a complex structure. Such assignments are termed as 'compound assignments.' Hence, in compound assignments, the process is carried out by first dealing with the required calculations using the general arithmetic type conversions. Once that has been dealt with, then the type conversion is performed.

Regardless, there are two particular scenarios out which a programmer will most likely face one during type conversions in assignments. These have been listed below:

1. In an assignment, if the variable's type is seen to be higher or bigger than the type of the value to which it is supposed to be assigned, then the value's type must be promoted to match the variable's type. For this type of conversion, we follow the rules that are defined in the Usual Arithmetic Types Conversion process. For example:

```
int i = 100;
long lg = i + 50;    // Result of type int is
                     // converted to long
```

2. In an assignment, if the type of the value is seen to be higher or bigger than the type of the variable to which it is being assigned to, instead, of promoting the variable's type, we must demote the value's type accordingly. Depending on individual circumstances, the following procedures cover most of the encounters in this type of conversion during assignments:

a. For converting an integral type to a smaller type, we must consider the two different cases for **signed** and **unsigned** int type conversions. Firstly, for converting an int type to a smaller type, we simply eliminate the most significant byte(s) from the bit pattern. If the resulting type is also unsigned, then the resulting bit pattern will be interpreted as unsigned. If the resulting type is signed, then the bit pattern will be simply interpreted as 'signed.' However, the original value of the number will only be retained if the new type is capable of representing it. An example of this process has been shown below:

```
long lg = 0x654321; short st;
st = lg;                    //0x4321 is assigned to st.
```

However, if we are converting an **unsigned** type into a **signed** type, then the original bit pattern remains the same as before, and this bit pattern is simply interpreted as ‘signed.’ For example

```
int i = –2; unsigned int ui = 2;
i = i * ui;
// First the value contained in i is converted to
// unsigned int (preserving the bit pattern) and
// multiplied by 2 (overflow!).
// While assigning the bit pattern the result
// is interpreted as an int value again,
// i.e. –4 is stored in i.
```

b. For converting a floating-point type to an integral type, we simply need to remove the decimal portion of the floating-point value. For instance, if we want to convert the floating-point number 2.9, then we simply remove the .96 part and round off the original number. Rounding off can be done by simply adding 0.5 to the floating-point number if it is positive and subtracting 0.5 from the floating-point number if it is negative. So, in the case of this example, by removing the decimal portion, we are left with the integer 2. However, after rounding it off (2.9 + 0.5), then the floating-point value is converted to an integer 3. However, the result of this type of conversion can be unpredictable at times, especially when the resulting value of **int** type is either too large or too small for the type itself. This is evident when converting a negative **floating-point** type value to an **unsigned** int type. For example

```
double db = –4.567;
int i; unsigned int ui;
i = db;            // Assigning –4.
i = db – 0.5;        // Assigning –5.
ui = db;           // –4 is incompatible with ui.
```

c. We will now discuss the case where a programmer wants to perform type conversion on a value belonging to the **floating-point** type into a smaller type. In such a type of conversion, there can be two results. One is that if the value of the **floating-point** number corresponds to the specified range of this destination type, then the original value of this number will be preserved at the cost of the number's accuracy. On the other hand, if the floating-point number is outside the range of the destination type (i.e., the value can be too large for the type to be able to represent it), then the end result will be unpredictable. An example of the conversion of a **floating-point** type to a smaller type has been shown below:

```
double d = 1.23456789012345;
float f;
f = d;                    // 1.234568 is assigned to f.
```

Some Other Type Conversions

Until now, we have chiefly discussed implicit type conversions, and while their use is more prevalent for fundamental C++ programming, there are other types of conversions as well that are worth discussing. In this section, we will talk about implicit type conversions in terms of function calls and a new type of conversion, which we haven't discussed up till now, Explicit type conversion.

Using Implicit Type Conversions in Function Calls

We have already discussed function calls in the 2nd chapter of this book, if there is something that you do not understand in this section about function calls, please refer to the topic that addresses this concept.

Implicit type conversion in a function call actually works pretty similarly to how implicit type conversions in assignments are processed. This is because, in function calls, the arithmetic types of the arguments being passed to the function are converted into the types specified in the function prototype. In this way, the parameters of the prototype function are followed and retained. For example:

```
void func( short, double);        // Prototype
int size = 1000;
// . . .
func( size, 68);                  // Call
```

In this example, you can see that the **func()** function has two arguments whose parameters have already been defined in the function prototype at the beginning. These arguments are supposed to be of **short** and **double** types. However, in the actual function call, you can see that the arguments being used are both **int** types. In such cases, implicit type conversion is performed to convert the **int** value **size** to a **double** type, and the integer 68 is converted to a **double** type value. Note that when an implicit type conversion takes place for this specific example, the compiler will issue a warning. The main purpose of this warning is to remind the user that since the **int** type is being converted to a **short** type, there are chances for data loss. To avoid such warnings, we have the option of performing an **explicit type conversion**.

Explicit Type Conversion

Just as the name suggests, in this type of conversion, we explicitly convert the expression types by leveraging the functionality of the **cast operator (type)**. In other words, we use the cast operator to perform an explicit type conversion. The syntax for this has been shown below

```
(type) expression
```

According to this syntax, we are directly converting the value assigned to the expression to a specified type. As such, explicit type conversion is also commonly referred to as '**casting**,' even the operator is named after this process.

Since the cast operator (type) is, by nature, a **unary operator**, it holds a higher level of precedence than the traditional arithmetic operators in C++. Let's understand explicit type conversions with the help of an example.

```
int a = 1, b = 4;
double x;
x = (double)a/b;
```

In this example, we can see that the value assigned to the variable **'a'** has been explicitly converted from an **int** type to a **double** type. Explicit type conversion wasn't performed on the other variable because it's type would automatically be converted to match the **double** type by the compiler through implicit type conversion. After the implicit type conversion is performed, the

floating-point division operation is then carried out by the compiler, and the exact result obtained is 0.25. This value is finally assigned to the **x** variable by using the assignment operator. Note that if we did not perform casting in this example, then the program would have performed a simple integer division operation, and the result it would have given would be 0, which is obviously incorrect.

The cast operator discussed in this section for explicit type conversion is for general purposes. In C++, there are other operators (for example, dynamic_cast<>) available as well that can be used for explicit type conversion, but those are used for specific cases only.

Chapter 7

The Use of References and Pointers in C++

In this chapter, we will learn about references and pointers, their use as parameters as well as the values returned by functions in programming. The discussion of references and pointers will mainly focus on passing by references and read-only access arguments.

Defining References

A variable or object that already exists in a specific location in the memory is given another name or alias, which is its reference. This variable can thus be accessed by using its original name or reference. Defining a reference for an object does not mean that it'll occupy extra memory in the program. The defined references execute operations along with the object to which it refers. A particular use of references is that they are used as parameters which process the result of functions and return their values.

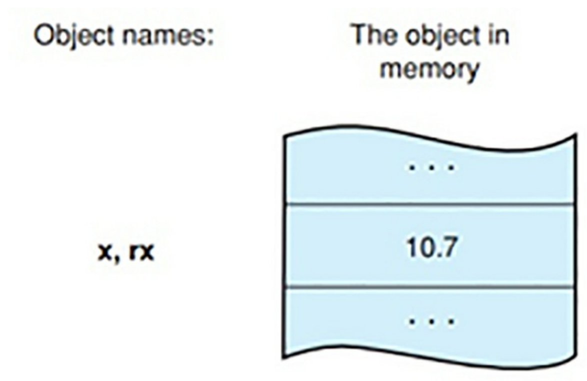
Defining References

References are denoted by the ampersand symbol &, and so T& would be the reference to type T.

Example:

```
float x = 10.7;

float & rx = x;  // or: float & rx = x;
```



Here it is clearly shown that rx is used as another way to express the variable x, and this type of reference is called the “reference to float.”

Example: --rx; // equivalent to --x;

In the above example, it is elaborated that the operations involved with rx will also affect the variable x. The ampersand character & used for indicating references is related only to declarations and is not the same as the one in the address operator that is &!. The address operator is used to evaluate and return the address of any object. If it is used as a combination with the reference, it ends up returning the address of the object with a reference.

Example: &rx // Address of x, thus is equal to &x

When defining the reference for an object, it must be initialized before the declaration and cannot be modified at a later stage. So, it is impossible to reuse a reference to define a different variable afterward.

Read-only references

If a constant variable is to be defined for a reference, a const keyword should be used so that the modification of the object by the reference can be avoided. However, a non-constant object can also have a constant keyword reference.

Example: int a; const int & cref = a; // ok!

The reference ‘cref ‘is a read-only identifier that gives read-only access to a variable such as “a” in the preceding example.

As compared to the normal references, a read-0nly identifier can be initialized by a constant.

Example: const double& pi = 3.1415927;

The constants do not occupy the memory, and hence, the temporary objects generated by the compilers are referenced.

References as Parameters

A variable or object that already exists in a specific location in the memory is given another name or alias, which is its reference. This variable can thus be accessed by using its original name or reference. Defining a reference for an object does not mean that it'll occupy extra memory in the program. The defined references execute operations along with the object to which it refers. A particular use of references is that they are used as parameters which process the result of functions and return their values.

References as Return Values

Returning references

When a function's return type is used as a reference type, the function call will represent an object and will act as an object as well.

Example:
```
string& message()          // Reference!
{
    static string str = " Today only cold cuts! ";
    return str;
}
```

In the function shown above, the reference is returned to a static string, which is not a normal auto variable present in the function message (). It would be a critical error to declare it as such because then the string would be destroyed after the program leaves the corresponding function, and the reference would then refer to an object that does not exist any longer. Thus, it is important to keep in mind that after leaving a function, the object in which the return value references must not be destroyed.

Calling a Reference Type Function

The function message () is defined as the type that implies a reference to string, and so string type object is represented by the calling message ().

Some valid statements with the function message () are:

```
message() = "Let's go to the beer garden!";
message() += " Cheers!";
cout << "Length: " << message().length();
```

Judging by the example given, the object which is referenced by the function call has the first new value assigned to it. After a new string is appended, the string is then written as an output in the third statement.

By defining the function type as a read-only reference, the modification of the object that is referenced can be prevented.

Example: `const string& message();  // Read-only!`

When the operators are overloaded, the type of reference chosen is the return type. An appropriately chosen function carries out the operations an operator executes when working with a user-defined type. Although overloading operators will not be discussed here, some examples of standard class operators will be provided.

Expressions with Reference Types

Example: Operator << of class ostream

```
cout << "Good morning" << '!';
```

All the expressions found in C++ are of a specific type, and if the expression is not void, they can give out a value as well. Expressions also belong to the reference types.

The Stream Class Shift Operators

Some operators can return the reference value to an object such as the << operator, which is used for stream input and the >> operator, which is specific to stream output.

Example: `cout << " Good morning "`

In this expression, the void types are not used, but instead, it is a reference to the object cout and represents that object. Hence, the << operator can be used on the expression repetitively.

```
cout << "Good morning" << '!'
```

This statement or expression is equal to:

```
(cout << " Good morning ") << '!'
```

By order of precedence, the expressions involving << operators are composed of the left.

Similarly to the << operator, the stream cin is represented by the expression cin >> variable and can also be used repeatedly.

Example: int a;   double x

```
cin >>  a >> x ;      // (cin >> a) >> x;
```

Other Reference Type Operators

The simple assignment operator = and compound assignments like += and *= are also used as reference type operators and the operand located on their left side receives the returned reference value.

Consider the following expression:

```
a = b or a += b
```

The variable “a” will be taken as the object and expression represent this object “a.” This can also be possible for the objects of a class type that is referred by an operator. But the available operators are specified by class definitions, such as the example where assignment operators = and += are defined in standard class string.

Example: string name ("Johnny ");

```
name += "Depp";          //Reference to name
```

It is possible to pass this expression as an argument to a function calling by reference because it is the type that represents an object.

Defining Pointers

```
// pointer1.cpp
// Prints the values and addresses of variables.
// ------------------------------------------------
#include <iostream>
using namespace std;
int var, *ptr; // Definition of variables var and ptr
int main() // Outputs the values and addresses
{ // of the variables var and ptr.
var = 100;
ptr = &var;
cout << " Value of var: " << var
<< " Address of var: " << &var
<< endl;
cout << " Value of ptr: " << ptr
<< " Address of ptr: " << &ptr
<< endl;
return 0;
}
```

The Output of the Sample Program

```
Value of var:        100     Address of var: 00456FD4
Value of ptr:  00456FD4     Address of ptr: 00456FD0
```

A program running efficiently usually does not manipulate the data and simply accesses the addresses of the data in the program's memory. Some examples include the linked lists and trees in which the elements are generated as they are needed during runtime.

Pointers

The concept of pointers is actually very simple and easy. By nature, pointers are variables, but they're not to be confused for any ordinary variable. These special variables have a value stored within them that points to an object while also detailing some of the object's features as well, such as its address and type. When we use a standard 'Address Pointer' (&) with an object, it creates something like a road map whose destination points to the object itself.

Example: &var   // Address of the object var

In the above example, we can see a pointer (&) being used with a variable 'var.' Since we are using a pointer, this allows us to create a virtual mind map for the program to refer to when finding the details of the ‘var’ variable, such as its address and its type.

Pointer Variables

The example shown above is using a pointer as a ‘constant.’ As we discussed before, pointers can be used as variables as well. Defining a pointer variable is actually pretty similar to defining a standard variable. The value that is assigned to such a variable is the memory address of a particular object we want to point to. An example has been shown below:

Example: int * ptr; // or : int * ptr;

In this example, we can see that we have a variable ‘ptr,’ which is of the **int** type. Notice the use of an asterisk as well. This particular character is special in defining pointer variables. When declaring this variable, the asterisk tells us that the ‘ptr’ variable points to an **int** type object. We call this asterisk as the ‘indirection operator.’

The type that can be assigned to a pointer variable also has a general form as well. This general form is ‘T*”. The character T refers to a data type (for instance, int, double, short, char, etc.), and you already know what the asterisk does.

Example: int a, *p, &r = a; // Definition of a, p, r

So when declaring a ‘pointer variable,’ we need to specify an address as well. The following example shows how to declare a pointer variable properly:

ptr = &var;.

References and Pointers

The similarity between references and pointers is that both refer to an object that is stored in the memory. Pointers differ in the aspect that they are not just an alias set for the object they reference but are individual objects with an identity of their own. Although the address for the pointer is already made, it can be changed by pointing it to a new address resulting in the pointer referencing another object.

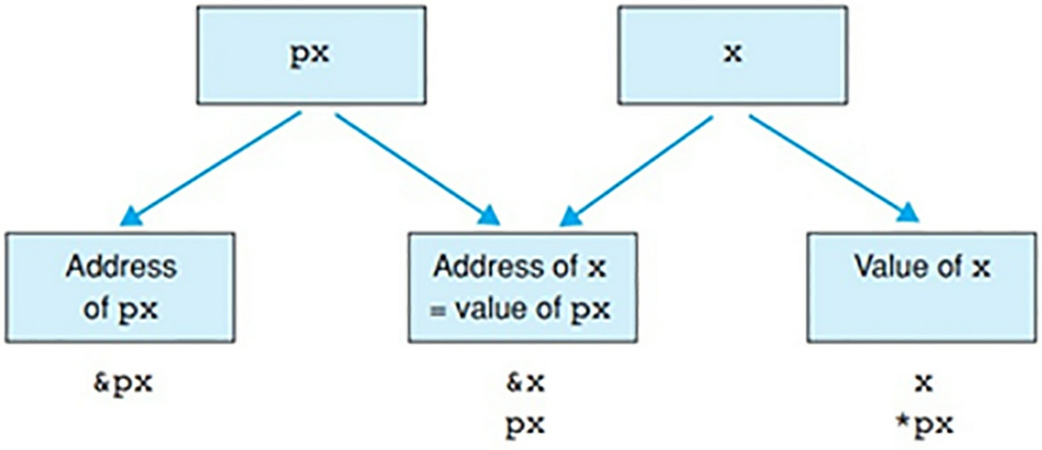
The Indirection Operator

Using the indirection operator

```
   double x, y, *px;

   px = &x;            // Let px point to x.
  *px = 12.3;          // Assign the value 12.3 to x
  *px += 4.5;          // Increment x by 4.5.
   y  = sin(*px);      // To assign sine of x to y.
```

Address and values of the variables x and px



Notes on Addresses in a Program

- The memory space that each pointer variable occupies does not depend on the type of object it refers to and is the same because it only stores the address of every object. The pointer variables on a 32-bit computer would typically occupy four bytes.

- Logic addresses are used in the program, which is allocated to physical addresses with the help of the system. In this way, the management of storage can be made efficient, and memory blocks not being used in the current time are swapped to the hard disk.

- When a valid address in C++ shows the value of 0, it indicates an error because no address has the value 0. In the standard header files, pointers use the symbolic constant NULL instead of 0, and these pointers are called NULL pointers.

Using Pointers to Access Objects

When a variable is pointing to an object, we can access this object by using the asterisk (the indirection operator). But we should not confuse the variable and the object with each other. ‘ptr’ is the variable, and ‘*ptr’ is the object.

Example:
```
long  a = 10, b,      // Definition of a, b
        *ptr;          // and pointer ptr.
ptr = &a;              // Let ptr point to a.
b = *ptr;
```

In the example shown above, we can see that the ‘ptr’ is pointing to a variable ‘a.’ Also, notice that in the beginning, the value of ‘a’ is being assigned to another variable ‘b.’ Hence, at the end, we can also substitute the variable ‘b’ with ‘a’ as they both represent the same thing, i.e., the object ‘a.’

```
long *ptr;
```

The above expression means that ptr is long* type, which is a pointer to long. Similarly, it can also be said that *ptr is a long type.

L-values

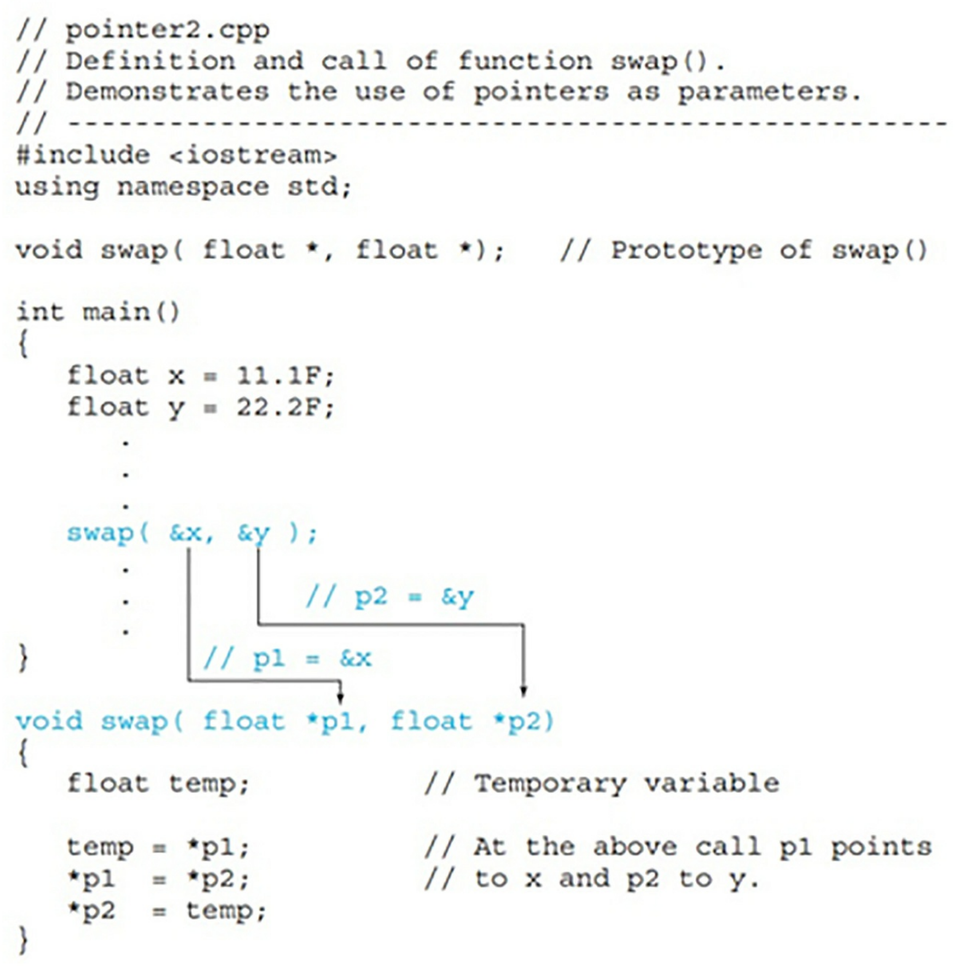
L-value in C++ is the type of expression that specifies a memory location to identify an object. The L-value is derived from an assignment operator and occurs in the compiler error messages. So, it is important that the left operand in the assignment = operator always specifies an address stored in the memory.

The expressions other than L-values, which cannot have a value assigned to it appear only on the right side of the assignment operator = and are termed as R-values.

In a statement, the variable would be the L-value, while constants and expressions such as x+1 are the R-values. L-values are also returned as results by using the indirection operators. Consider a pointer variable “p,” then p and *p would both be L-values because *p is the object that variable “p” points to.

Pointers as Parameters

The following program demonstrated shows the implementation of pointers as parameters.



```
// pointer2.cpp
// Definition and call of function swap().
// Demonstrates the use of pointers as parameters.
// -----------------------------------------------------
#include <iostream>
using namespace std;

void swap( float *, float *);   // Prototype of swap()

int main()
{
   float x = 11.1F;
   float y = 22.2F;
       .
       .
       .
   swap( &x, &y );
       .
       .              // p2 = &y
       .
}             // p1 = &x

void swap( float *p1, float *p2)
{
   float temp;          // Temporary variable

   temp = *p1;          // At the above call p1 points
   *p1  = *p2;          // to x and p2 to y.
   *p2  = temp;
}
```

Objects as Arguments

When a function is called, and an object is passed over as an argument to the required function, the possible situation that may occur would be:

- The function parameter is of the same type as the object which was passed to it as the argument. Thus, the function that is called receives a copy of the object (passing by value).
- The function parameter is a reference that means the said parameter is an alias for the argument. The function that is called then manipulates the object which was passed to it by the calling function (passing by reference).

In the passing by value, it is clear that the function that was passed over the argument cannot manipulate it, but it is possible when using a ‘passing by reference.’ On the other hand, a third situation related to passing by reference is passing the pointers to the required function.

Pointers as Arguments

When a function parameter is declared as a pointer variable, it is possible to declare a function in a way that an address can be passed as an argument to the function.

For example, by using the statement:

Example: long func(int *iPtr)

```
{
    // Function block
}
```

The parameter iPtr is declared as an int pointer, so the address of an int value can be passed to the function func () as an argument.

A function can access and manipulate an object with the help of the indirection operator only if it knows the memory address of that object.

The function swap () shown in the sample program is used to swap the values given by the variables x and y within the calling function. The addresses of the variables &x and &y are already passed to the function as arguments, which enables the function to access these variables and manipulate them.

The swap function () contains two parameters “p1” and “p2” which are declared as float pointers. These pointers are initialized with the addresses of variables x or y by the given statement.

```
swap ( &x, &y );
```

By manipulating the expression *p1 and *p2, the function can access variable x and y available in the calling function. In this way, their values can be exchanged.

Chapter 8

The Basics of File Input and File Output in C++

In the first few chapters of this book, we briefly discussed the **iostream** library, also known as the library, to feature sequential file access stream classes. In this chapter, we will build our discussion on this fundamental concept and explore it in more detail and depth. Understanding file streams are very important in programming as they are a gateway for practicing portable file handling techniques. Moreover, file operations are one of the very foundational tasks on which C++ programs are built upon. Just as how functions, classes, objects, and variables all come together to make up the basic structure of a program, file streams provide a structural representation of storing data within a program to an external storage device. This is a very important process because a program primarily stores its data in the volatile system memory. Hence, when we close a program, the data in this volatile storage is immediately lost. That's why we need certain techniques to not only output data from a program to permanent storage but also input data into the program as well.

The Basic Concept of Files

Before we discuss file streams, we must first understand the concepts of file operations and file positions.

File Operations

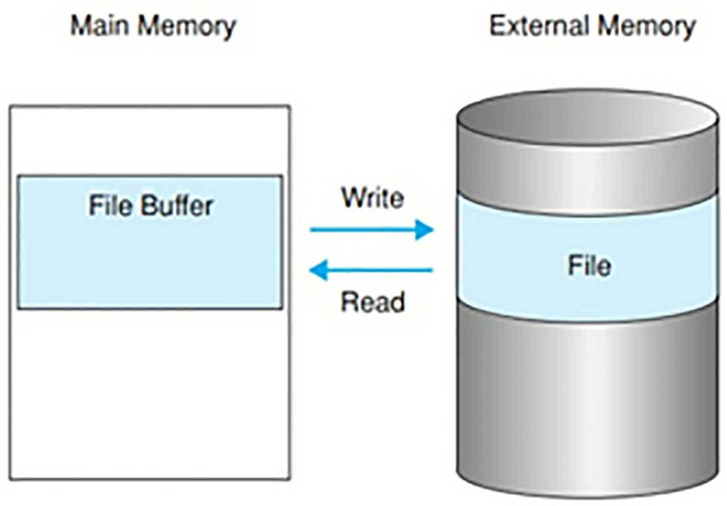
Just as how lone characters or an entire string of characters can be displayed on a screen as an output, such characters can also be written to a text file as well as representing this data. Now let's talk a bit about records. Record is an umbrella term for referring to a file that houses data forming logical units. Generally, records are stored in files. This is achieved by the **write**

operation, which handles the process of virtually storing a specified record to a specified file. If the file already has an existing record, the write operation either updates the already present record or simply adds in a new record to the file alongside the pre-existing one. When we want to access the contents of the record stored in a file, we are issuing a **read** command which takes the contents of the record and copies it to the program's defined data structure.

Similarly, objects can also be stored in the permanent storage of the system instead of the volatile storage. However, the process isn't entirely the same as storing and reading record files, as we need to do more than just store the internal data of the object itself. When storing objects, we need to make sure that the object, along with its data, is accurately reconstructed when we issue a read command. For this purpose, we not only need to store the object's type information, but we also need to store the included references to other objects as well.

It is important to keep in mind that all of the external storage devices (for instance, a hard disk) have a block-oriented storage structure. This block-oriented nature of storage means that the data is stored into the device in blocks, and the sizes of these blocks are always multiples of 512 bytes.

Efficient and simple file management simply refers to the concept of taking the data you need to store and transferring it to the temporary storage of the main memory, which is also known as **'file buffer.'** Here's a visual representation of this concept.



File Positions

In a C++ program, a file is interpreted as a large array of bytes. Keeping this in mind, the structural elements of this file is solely the responsibility of the programmer to handle. How much flexibility the file's structure provides the programmer is dependent on how he structured the file itself.

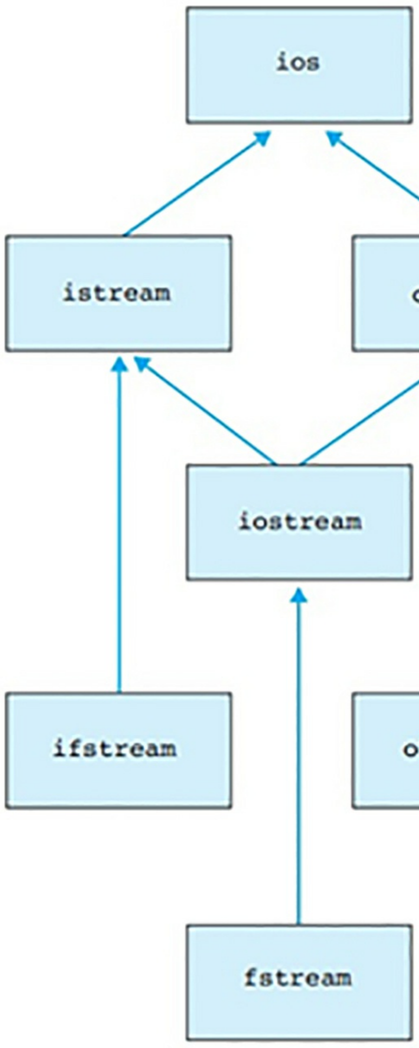
In a file, there are many characters represented by bytes. Each byte in a file holds a specific position. For instance, the first byte of the file will be assigned the position 0, the next byte will be assigned the position 1 and this trend continues. As a consequence of this feature of files, a very important term arises, which is known as 'current file position.' This refers to the position that will be assigned to the upcoming byte, which is going to be written or read from the file. So, whenever a new byte is added to a file, the current file position is displaced positively (increased) by the value of 1.

In sequential access, things are a little different. As we know that the word 'sequential' means 'in a sequence or following a sequence,' the data that is being written or read to file is done in a fixed and defined order. In other words, there is a defined sequence in which the programmer can read or write data, and it is not possible to deviate from this sequence. So if you execute the very first read operation on a file, the program will begin reading the file from the very beginning. This means that if you want to access a specific piece of information located within the file, you will have to go through the entire sequence, i.e., start from the beginning and scrolling through the contents of the file until you find what you’re looking for. In terms of write operations, they have a little more freedom compared to read operations as they can easily create new files, overwrite existing files, or add new data to an existing file.

In contrast, random file access means that we can access or read any part of the file without having to follow through a sequence at any given time. This allows for instantaneous access to the contents of the file. The concept of providing easy access to files refers to this technique of random access, and programmers have the freedom of specifying current file positions per their needs.

File Stream Classes

In C++, there are several classes standardized for file management purposes. These classes are commonly referred to as ‘file stream classes’ and offer easy file handling functionality for the program. Here’s a flow-chart highlighting some of the most commonly used file stream classes and their hierarchy relative to each other.



Although, as a programmer, you need to consider the file management classes and implement them properly, what you don't need to worry about regarding file management during programming is buffer management or even the system specifics.

In C++, the major file stream classes have already been standardized, allowing programmers the freedom and capability of developing portable C++ programs. By portable, we do not mean easy to carry around like a laptop or a smartphone. Portable in programming means that this particular program can be easily ported over to other platforms such as Windows or Unix. Hence, standardized file stream classes make it easier for programmers to develop programs that can be easily ported to other platforms. All it takes is a simple recompilation of the program for each platform it's being used on.

File Stream Classes Belonging to the iostream Library

If you refer to the flow-chart shown in the previous section highlighting the hierarchy of the file stream classes, we can see that this family of classes have something known as 'base classes.' A base class is a class from which other classes are derived. In other words, we can understand base classes by remembering them as 'parent classes' from which other classes can be created; however, 'base class' is the official programming term. We have already used some of these base classes in the programs demonstrated in this book as well. Here's an explanation of what class has been derived from which parent class:

- **istream** is the parent class for the **ifstream** class. In other words, the **ifstream** class has been derived from the **istream** class. The purpose of the **ifstream** class is to allow the operations of file reading.

- The **ostream** class is the parent class for **ofstream**. In other words, the **ofstream** class is created from the **ostream** class, and its purpose is to support writing operations for files.

- The **iostream** class is the parent class for **fstream**. In other words, the **fstream** class is created from the **iostream** class, and its purpose is to support operations such as reading and writing for files.

So an object that is part of the **file stream** class is referred to as a **'file stream**.' The standardized file stream classes are defined in a header file known as **fstream,** and to use these classes in a program, we must add this header file into the program using the **#include** directive.

Functionalities of the File Stream Classes

Whenever we create a file stream class from a base class, the newly derived class inherits all of the functionalities of its parent class as well. In this way, class functionalities such as methods, operators, and even manipulators for **cin** and **cout** are available to these classes as well. Hence, every file stream class comes with the following functional features:

- Methods that have been defined for operations such as non-formatted reading or writing specifically for single characters and data blocks.

- Operators ('<<' and '>>') that are used for formatted read and write operations to or from files.
- The methods and manipulators that have been defined for formatting character sequences.
- The methods that have been defined for tasks such as state queries.

Creating Files through a C++ Program

Let's see a program demonstrating the creation of file streams and then break it down and understand the fundamentals of this concept.

```
// showfile.cpp
// Reads a text file and outputs it in pages,
// i.e. 20 lines per page.
// Call: showfile filename
// ----------------------------------------------------
#include <iostream>
#include <fstream>
using namespace std;
int main( int argc, char *argv[])
{
   if( argc != 2 )                    // File declared?
   {
     cerr << "Use: showfile filename" << endl;
     return 1;
   }
   ifstream file( argv[1]);               // Create a file stream
                                         // and open for reading.
   if( !file )                        // Get status.
   {
     cerr << "An error occurred when opening the file "
          << argv[1] << endl;
     return 2;
   }
```

```
    char line[80];
     int cnt = 0;
     while( file.getline( line, 80))              // Copy the file
     {                                    // to standard
       cout << line << endl;                   // output.
       if( ++cnt == 20)
       {
          cnt = 0;
          cout << "\n\t ---- <return> to continue ---- "
               << endl;
           cin.sync(); cin.get();
          }
       }
     if( !file.eof() ) // End-of-file occurred?
     {
       cerr << "Error reading the file "
           << argv[1] << endl;
       return 3;
     }
     return 0;
  }
```

Opening a File

Before we can proceed to manipulate a file in a program, we first need to open and access it. To open a file, we need to perform two fundamental actions:

- State the name of the file along with its directory or path where it is located on the system's permanent storage.
- Define a **file access mode** that corresponds to the file we want to open.

If the file we want to open does not have a directory or a path that is stated explicitly, it means that the file should be in the current directory of the program. The file access mode specifies the read and write permissions granted to the user for the file. This means that if the file access mode is defined as read-only, then we can only access the contents of the file but cannot modify it. When a program is terminated, all the open files that are associated with it are closed as well.

Defining the File Stream

When we create a file stream, we can also open the file at the same time as well. To do so, all we have to do is state the file’s designated name. The program demonstrated at the beginning of this section uses default values for defining the file access mode.

```
        ifstream myfile("test.fle");
```

In this statement, since we have not specified any particular directory or path for the file ‘test.file,’ this tells the program that the file must be located in the same directory. The file is opened by the constructor of the **ifstream** class to perform a read operation. Once a file has been opened in a program, the current file position is specified at the start of the file.

Also note that if you specify a write-only file mode access, then it’s no longer necessary for the actual file to even exist in the system. If there is no file corresponding to the file name for write-only access mode, then the program will create a new file with this specified name. However, if a file with the name specified actually exists, then the write-only access mode will delete this file.

In this line of code

```
        ofstream yourfile("new.fle");
```

We are creating a new file with the name of ‘new.fle.’ Once this file has been created, the program opens the file to perform the write function. Just as we recently discussed, if the directory has an existing file with the same name, then it will be first deleted before the new file is created.

We can also create a file stream which does not necessarily refer to any particular file. This file can be opened later by using the **open()** method. For example

```
        ofstream yourfile;
        yourfile.open("new.fle");
```

These two statements perform the same task as the “ofstream yourfile("new.fle");” line. To elaborate, the open() method opens the file by using the same set of values, which are also used by the default constructor for the file stream class.

Usually, fixed file names aren’t always used by experienced programmers in every instance. If we analyze the program shown at the beginning of this section, we will see that the name of the file we want to manipulate is stated in the command line instead. If we do not provide a suitable file name for the program to operate on, then it will simply generate an error message and close. Another alternative route of defining file names is to leverage the interactive user input feature in programs.

Modes when Opening Files

In this section, we will discuss open modes that can be used with constructors as well as the **open()** method. But before we dive into this concept, let’s first understand the different flags for the open mode of a file. A table showing the open mode flags along with their corresponding functions have been displayed below:

Flag	**Function**
ios::in	Opens an existing file for input
ios::out	Opens a file for output. This flag implies ios::trunc if it is not combined with one of the flags ios::in or ios::app or ios::ate
ios::app	Opens a file for output at the end-of-file
ios::trunc	An existing file is truncated to zero length
ios::ate	Open and seek to end immediately after opening. Without this flag, the starting position after opening is always at the beginning of the file
ios::binary	Perform input and output in binary mode.

(Explanation of the flags referenced from ‘A Complete Guide To Programming in C++ by Ulla Kirch-Prinz and Peter Prinz)

It is important to note that all of the flags mentioned above are already defined in the **ios** base class (which is a parent class to all other file stream classes). Moreover, all of these file stream classes are of the **ios::openmode** type.

The Default Settings of an Opened File

Whenever we open a file, the default values used by the constructor of the file stream class and the **open()** method are:

Class	**Flags**
Ifstream	ios::in
Ofstream	ios::out | ios::trunc
Fstream	ios::in | ios::out

If you want to open a file without the default values assigned to its constructor and open() method, then we will need to supply the program with two things; the file name and the open mode. This is an absolute requirement to open a file that already exists in the directory in a write-only access mode without deleting the original file.

Understanding the Open Mode Flags

We can pass an additional argument to the open mode alongside the file name to both the constructor of the file stream class as well as the open() method. This is because the open mode of the file is dependent on flags.

In programming, a flag is represented by a single bit. If the flag is raised, then it will have a ‘1’ value, and if the flag is not raised, then it will have a ‘0’ value.

Another important element in flags is the bit operator “|.” This operator is commonly used to combine different flags. However, in all of the cases, one of the two flags, ‘ios::in’ or ‘ios::out,’ should be stated. This is because if the ios::in flag is raised when opening the program, it tells the system that the file already exists and vice versa. If we do not use the ios::in flag when opening a file, the program will create this file if it doesn’t exist in the directory.

In the following statement:

```
fstream addresses("Address.fle", ios::out | ios::app);
```

We are opening a file by the name of ‘Address.fle,’ and if it does not exist within the directory, then it will be created. According to the flags and file stream class being used in this statement, the file is being opened for a write operation at the end of the file. After the completion of each write operation, the file will grow automatically.

There’s also an update mode which allows us to open a file to append data into its existing contents or update the existing data and is often used in random file access mode. This is done by using the default mode for the **fstream** class, which is (ios::in | ios::out), allowing the user to open a file that already exists for read and write operations.

Error Handling

Encountering errors when opening files is a common phenomenon. This can be due to various reasons, with the most common ones being that you either don’t have the required privileges to access the file or the file simply does not exist. To handle any error that may occur, we implement a flag **failbit** that monitors the state of the operation. This flag is from the base class **ios,** and if an error does occur, this flag is raised. Querying the flag is also very simple. We can either use the **fail()** method to directly query the flag or check the status of the file stream by using the **if** conditional to query the flag indirectly. For example

```
if( !myfile)          // or: if( myfile.fail())
```

The failbit flag is also raised when an error occurs in the read or write operations. However, not every error indicates a critical malfunction or of some sort. For instance, if the program encounters a read error, then this may mean that the program has read through all the file’s contents, and there’s nothing left to read. In such cases, we identify the nature of the read error properly by using an end-of-file method, which is actually referred to as **eof()**. We can separate such types of normal read errors from other types of hindering read errors by querying the **eof** bit as shown below:

```
if( myfile.eof())       // At end-of-file?
```

Closing Files

It is recommended that whenever we are done working with files or completed the file manipulation tasks, we must always close the files. The effectiveness of this practice is widely supported due to two main reasons:

- If a program is not properly terminated, then the file opened by the program may experience data loss.
- A program is not capable of opening numerous files at the same time; there's a limit to the files that can be opened simultaneously. As such, we should properly close the files we are not working with to avoid any errors in the program.

A program that is properly terminated will automatically close any open files that are associated with it. However, there are cases where a program is not properly terminated. To avoid those unforeseeable cases, it is important always to close the files directly once they are not being used.

Here's a program that demonstrates the concepts discussed up till now.

```
// fcopy1.cpp : Copies files.
// Call: fcopy1 source [ destination ]
// ---------------------------------------------------
#include <iostream>
#include <fstream>
using namespace std;
inline void openerror( const char *file)
{
  cerr << "Error on opening the file " << file << endl;
  exit(1); // Ends program closing
}                                  // all opened files.
void copy( istream& is, ostream& os);         // Prototype
int main(int argc, char *argv[])
{
  if( argc < 2 || argc > 3)
  {  cerr << "Call: fcopy1 source [ destination ]"
```

```
              << endl;
           return 1;                          // or: exit(1);
        }
        ifstream infile(argv[1]);                  // Open 1st file
        if( !infile.is_open())
           openerror( argv[1]);
        if( argc == 2)                         // Just one sourcefile.
           copy( infile, cout);
        else                                // Source and destination
        {
           ofstream outfile(argv[2]);                 // Open 2nd file
           if( !outfile.is_open() )
              openerror( argv[2]);
           copy( infile, outfile);
           outfile.close();                         // Unnecessary.
         }
         infile.close();                          // Unnecessary.
         return 0;
      }
      void copy( istream& is, ostream& os)        // Copy it to os.
      {
        char c;
        while( is.get(c) )
           os.put(c);                          // or: os << c ;
      }
```

The close() And is_open() Methods

Notice that each of the file stream classes used in the program demonstrated a method defined as a **void** type. This is the **close()** method, and as its name suggests, it’s purpose is to terminate the file which is occupied by the stream in which the method is used. For example:

```
myfile.close();
```

Even though the file on the specified file stream is terminated, the file stream itself is left untouched. This means that by closing a file on a particular file stream, we can open another file immediately on the same stream. To check whether a file is currently occupying a file stream, we use the **is_open()**

method to do so. For instance,

```
        if( myfile.is_open() )
        { /* . . . */ }             // File is open
```

The exit() Function

When using the global **exit()** function, files that are open and being accessed by the program are closed. The main reason for using this global terminating function as opposed to the **close()** method is that we are not only closing the open files, but we are also terminating the program itself as well. In this way, the program is properly closed, and a **status** error code is returned to the corresponding calling process. The prototype of the **exit()** function is shown below:

```
        void exit( int status );
```

The reason for returning a **status** error code to the calling process is because the calling process evaluates the **statu**s. Usually, the calling process in such cases is the command-line interpreter itself, for example, Unix shell. When a program is successfully terminated without any problems, it returns an error code ‘o’. Similarly, in the **main()** function, the two statements **return n** and **exit(n)** are equivalent to each other.

In the program shown in the next section, you will see a program that is instructed to copy the contents of a file to and paste it to a destination file. The file is stated in the command line, and the program proceeds to copy it. If the user does not specify the destination file, then the original file is simply copied to the program’s standard output.

Read and Write Operation on Blocks

All of the file stream classes are capable of utilizing the **public** operations that have been originally defined in their parent classes, otherwise known as **base classes**. Hence, by using appropriate file stream classes for a program, we can easily perform write operations for transferring formatted or unformatted data to a specified file. Similarly, we can also perform a read operation to go through the data contents of the file in either entire blocks at a time or one character at a time.

Here’s a program demonstrating the use of read and write operations for blocks of data.

```
// Pizza_W.cpp
// Demonstrating output of records block by block.
// ---------------------------------------------------
#include <iostream>
#include <fstream>
using namespace std;
char header[] =
" * * * P I Z Z A P R O N T O * * *\n\n";
// Record structure:
struct Pizza { char name[32]; float price; };
const int MAXCNT = 10;
Pizza pizzaMenu[MAXCNT] =
{
   { "Pepperoni", 9.90F }, { "White Pizza", 15.90F },
   { "Ham Pizza", 12.50F }, { "Calzone", 14.90F } };
int cnt = 4;
char pizzaFile[256] = "pizza.fle";
int main()                    // To write records.
{
   cout << header << endl;
   // To write data into the file:
   int exitCode = 0;
   ofstream outFile( pizzaFile, ios::out|ios::binary );
   if( !outFile)
   {
      cerr << "Error opening the file!" << endl;
      exitCode = 1;
   }
   else
  {
      for( int i = 0; i < cnt; ++i)
        if( !outFile.write( (char*)&pizzaMenu[i],
                              sizeof(Pizza)) )
        { cerr << "Error writing!" << endl;
         exitCode = 2;
```

```
            }
        }
        if( exitCode == 0)
         cout << "\nData has been added to file "
               << pizzaFile << "\n" << endl;
        return exitCode;
    }
```

Formatted and Unformatted Input and Output

In the programs demonstrated up until now in this chapter, we have seen the use of some important methods **get()**, **getline()** and **put()** to instruct the program to perform read or write operations to and from text files. Data that is formatted, such as numerical values, require the '<<' and '>>' operators for input and output. In addition, we also need specific formatting methods and proper manipulators to handle formatted data. For example:

```
double price = 12.34;
ofstream textFile("Test.txt");
textFile << "Price: " << price << "Dollar" << endl;
```

In these lines of code, we can understand that the actual **test.txt** file itself will have a line that will correspond to "Price .." and this line will match exactly with the output shown on the screen.

Transferring Blocks of Data

Transferring entire data blocks is mostly done by issuing a write operation through the **write()** method. This method belongs to the **ostream** class and transfers the number of bytes specified by the user from the system's main memory to the destination file. The prototype of this method has been shown below:

```
ostream& write( const char *buf, int n);
```

Since the **write()** method gives a reference value to the corresponding file stream, we can use this to check if the write operation completed successfully or if it wasn't able to transfer the total bytes of data completely. The following statements show how this can be done:

```
if( ! fileStream.write("An example ", 2) )
```

```
cerr << "Error in writing!" << endl;
```

When the program tries to perform a write operation to transfer the first two characters, “An,” then it will issue a warning if the write operation encounters an error; otherwise, everything will go smoothly.

We can also perform a read operation by using the **read()** method belonging to the **istream** class to read the blocks of data within a specified file. When a read operation is performed, the **read()** method takes a data block from the source file and transfers it to the buffer memory of the program to read it. Once the data block has been read, the buffer memory of the program is cleared, and the next data block is transferred until we reach the end-of-line of the file. The prototype method of the read operation is shown below:

```
istream& read( char *buf, int n);
```

It is important to note and remember that the **read()** and **write()** methods are primarily used with records that are of a fixed length. Moreover, the data block we want to transfer can have more than one record as well. Last but not least, the buffer in the system’s main memory can have two possible structures: a simple structure variable or an entire array whose elements are part of the structure type itself. When accessing the main memory for a data block, we must specify the address referring to the specific area of memory in which the data block is found to the argument **(char *)**. The implementation of the **read()** and **write()** methods have been demonstrated in the following program:

Creating a Class Account

```
// Class Account with methods read() and write()
// ---------------------------------------------------
class Account
{
   private:
     string name;          // Account holder
     unsigned long nr;      // Account number
     double balance;        // Balance of account
   public:
     . . .      // Constructors, destructor,
                // access methods, ...
     ostream& Account::write(ostream& os) const;
     istream& Account::read(istream& is)
};
```

Implementing the read() and write() methods.

```
// write() outputs an account into the given stream os.
// Returns: The given stream.
ostream& Account::write(ostream& os) const
{
   os << name << '\0'; // To write a string
   os.write((char*)&nr, sizeof(nr) );
   os.write((char*)&balance, sizeof(balance) );
   return os;
}
// read() is the opposite function of write().
// read() inputs an account from the stream is
// and writes it into the members of the current object
istream& Account::read(istream& is)
{
   getline( is, name, '\0'); // Read a string
   is.read( (char*)&nr, sizeof(nr) );
   is.read( (char*)&balance, sizeof(balance));
   return is;
}
```

Conclusion

It’s a delight that you have finally reached the end of this book, and we hope that you had as much fun reading the book as we did writing the book. We hope that you followed the guidelines properly on how to approach each chapter in this book and have learned a lot. The starting chapters of this book have laid the foundation for upcoming and technical concepts such as performing arithmetic type conversions, handling input and output file streams, using references and pointers, and controlling the flow of C++ programs using control-flow operators. If you have properly understood and digested the information laid out in the starting chapters, then the relatively complicated topics should also be understandable.

While journeying through the chapters, we have come across some very basic concepts and some that act as a base for the concepts in the intermediate and advanced level of programming. However, challenging oneself in learning and polishing your skills for an even higher skill cap in programming is what defines a programmer. Relentless and stubborn efforts are the key qualities that enable programmers to create state-of-the-art programs and innovate with their brilliant ideas. Aspiring for such goals should be your main concern when learning to program as it will not only bring you to even bigger heights but also open a world of infinite possibilities in the realm of computers. It is our deeply sought after wish that this book was up to the standards of the reader and that this book served as the perfect starting point for the reader’s journey in programming.

References

1). A Complete guide to programming in C++ by author: **Ulla Kirch-Prinz & Peter Prinz.**

C++

Advanced Guide to Learn C++ Programming Effectively

BENJAMIN SMITH

Introduction

I want to thank you for choosing this book, *‘C++ - Advanced Guide to Learn C++ Programming Effectively,’* and I hope you find the book informative.

If you have read the previous book, you have gathered a basic idea of some concepts in C++ and how you can use loops and conditional statements to address different problems. This, however, does not mean you have mastered the art of programming in C++. You need to have more information to help you write robust programs and applications. This book will shed some light on some advanced topics in C++, which will enhance your understanding of C++.

The book will shed some light on the references and pointers in C++ and their importance. It also provides information on data structures and how you can use them in C++. Since object-oriented programming (OOP) is an important concept in most programming languages, this book sheds some light on what it is and the various concepts in OOP.

In this book, you will learn more about how you can optimize the performance of your code. When you write any code, you need to test it to determine if it runs correctly. You need to find the errors in your code and find a way to overcome those errors. So, what are you waiting for? Grab a copy of this book now and get started. By the end of the book, you will learn how to write code and improve it, so there are no errors and issues when you compile the code.

Chapter 1

Using Pointers in C++

Pointers make it easier to perform specific types of tasks in C++. They are easy to use, and it is best to use them to perform activities or tasks, such as dynamic memory allocation. We have looked at the basics of memory allocation in the previous book. This chapter will shed some light on how best you can use pointers in C++.

Every variable you enter into a program or code will be stored in a memory location. Each location has its own address, and these addresses can be accessed in the code using the '&' operator. This operator denotes that section in the memory where the variable is stored. Let us look at the following example to see how you can print the location or every variable defined in the code.

```
#include <iostream>
using namespace std;
int main () {
   int  var1;
   char var2[10];
   cout << "Address of var1 variable: ";
   cout << &var1 << endl;
   cout << "Address of var2 variable: ";
   cout << &var2 << endl;
   return 0;
}
```

When you compile the code written above, you obtain the following output:

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

The terms 0xbfebd5c0 and 0xbfebd5b6 are the locations in the memory where these variables are stored.

Introduction to Pointers

Before we look at how you can use pointers, let us first understand what a pointer is. Pointers are variables that take the address of a different variable in the code. The syntax of a pointer is as follows:

```
type *var-name;
```

The keyword type in the above syntax is the data or base type of the pointer. Make sure the type is a valid data type in C++. The value var-name is the pointer’s name. You need to use the asterisk in the syntax when you define the pointer. C++ throws an error if you forget to use it. The following are some methods to define pointers.

//The following statements are used to define or declare integer, double, float, and character pointers.

```
int    *ip;
double *dp;
float  *fp;
char   *ch;
```

Pointers will only take hexadecimal values since they only take the values of the variables you point them to. You can define a pointer as an integer, double, character, string, etc., but it only represents an address in the memory. The only difference is that when you assign a data type to a pointer when you define it, you indicate to the compiler that you are pointing to a variable with the same data type.

How to Use Pointers in C++

You can perform different operations in C++ using pointers:

1. Defining a pointer variable
2. Assigning the pointer with a variable whose address it stores
3. Accessing the value present in the memory location stored in the

pointer

You can perform these operations using the operator ‘*’ which indicates to the compiler that it needs to return the value of the variable stored at the memory location or address indicated by the pointer. The following example uses these operations:

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

When you run the code and compile it, you obtain the following output:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Types of Pointers

It is easy to understand how you can use pointers in C++. Having said that, if you make mistakes when you use them in your code, you will receive multiple errors. The following are some concepts to bear in mind when it comes to pointers:

S. No.	**Concept**	**Description**
1	Null Pointers	You can use null pointers in C++. This pointer is a constant variable that has a value of zero defined in numerous libraries used in C++.
2	Pointer Arithmetic	You can use the following operators on pointers: 1. ++ 2. + 3. – 4. - -
3	Pointer vs. arrays	There is a very close relationship between arrays and pointers.
4	Arrays of pointers	If you do not want to introduce numerous variables for pointers, you can create an array to store the same data type pointers.
5	Pointer to pointer	C++ allows you to use one pointer to indicate to another pointer.
6	Passing a pointer as an argument in a function	You can pass pointers as arguments in functions using either a reference or address. These allow the compiler to pass the pointer as the argument in the function.
7	Returning pointers from functions	You can use a function to indicate a local variable to store the value of the pointer. You can use: 1. A static variable 2. A local variable

		3. Dynamically allocated memory

Chapter 2

References in C++

Unlike pointers, references are used as aliases in C++. a reference is used to refer to a variable present in the existing code. When you initialize a reference and assign it to a variable in the code, you can use the variable itself or the reference variable to call the value stored in the variable if you need to use it in a different function.

Difference Between References and Pointers

People often confuse themselves when it comes to references and pointers. There are three differences between the two:

1. As mentioned in the previous chapter, you can have a null pointer, but you cannot have a null reference in your code. Make sure the reference is always tagged to a variable or function which has a return value.
2. When you initialize and assign a reference to a specific object, you cannot change its value to another object in the code at any point. You can use pointers to look at different objects at varied points in the code.
3. Every reference needs to be initialized before it is tagged to any variable. Unlike pointers, you cannot initialize a reference in any line of the code.

How to Create References

From the first book, you know that every variable has a name. Let us assume that this name is the label attached to the location of the variable’s value in the memory. When you tag a reference to the variable, it becomes the second label attached to the location. Therefore, you can refer to the value in the memory location using either the reference or the original variable name. Let us consider the following example:

```
// Initialize a variable 'i' and assign it a value
int i = 4;
float j = 2.8;
// Declare a reference variable in your code for the above variable
int &r1 = i;
float &r2 = j;
```

The ampersand (&) in the above line is your reference. Read the above two lines of code as follows:

1. The integer reference, r1, has been initialized and tagged to the variable i
2. The integer reference, r2, has been initialized and tagged to the variable j

In the following example, we look at how you can use references on variables with the data types double and int.

```
#include <iostream>
using namespace std;
int main () {
   // The following statements are used to declare the simple variables
in the code
   int    i;
   double d;
     // The following statements are used to declare and assign the
reference variables to the simple variables
   int&    r = i;
   double& s = d;
    i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;
    d = 11.7;
   cout << "Value of d : " << d << endl;
```

```
    cout << "Value of d reference : " << s  << endl;
     return 0;
}
```

When you compile the above code, you will obtain the following output:

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

Coders often use references as function return values or argument lists. The following are two points to bear in mind when you write code in C++:

S. No.	**Concept**	**Description**
1	Using references as function parameters	You can pass references as parameters in functions. It is safer to use them as parameters instead of using simple variables
2	Using references as function values	References can be used like other parameters or data types as return values

Chapter 3

Introduction to Data Structures in C++

C++ allows you to use different variables and structures, such as arrays and lists. We have looked at these in brief in the first book. This chapter introduces the different ways you can use these data structures to perform different activities in C++. You can use arrays to define different variables or combine different elements across the program or code into one variable, as long as they fall into the same category. A structure, however, allows you to combine different variables and data types. You can use a structure to define or represent records. Let us assume you want to track the books on your bookshelf. You can use a structure to track various attributes of every book on your shelf, such as:

1. Book ID
2. Book title
3. Genre
4. Author

The Struct Statement

You need to use the struct statement to define a structure in your code. This statement allows you to develop or define a new data type for your code. You can also define the number of elements or members in the code. The syntax of this statement is as follows:

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

It is not mandatory to use the structure tag when you use the statement. When you define a member in the structure, you can use the variable definition method we discussed in the previous book. For instance, you can use the method int i to define an integer variable. The section before the semicolon in the struct syntax is also optional, but this is where you define the structure variables you want to use. Continuing with the example above, let us look at how you can define a book structure.

```
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
} book;
```

How to Access Members

Once you define the structure, you can access it using a full stop, which is also called the member access operator. This operator is used as a period or break between the structure member and the variable name. Make sure to enter the variable name you want to access. You can define the variable of the entire structure using the struct keyword. Let us look at an example of how you can use structures:

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
};
int main() {
   struct Books Book1; // This is where you declare the variable Book1
in the Book structure
```

```
   struct Books Book2; // This is where you declare the variable Book2
in the Book structure
// Let us now look at how you can specify the details of the first
variable
   Book1.book_id = 120000;
   strcpy( Book1.book_title, "Harry Potter and the Philosopher’s
Stone");
   strcpy( Book1.genre, "Fiction");
   strcpy( Book1.author, "JK Rowling");
// Let us now look at how you can specify the details of the second
variable
   Book2.book_id = 130000;
   strcpy( Book2.book_title, "Harry Potter and the Chamber of
Secrets");
   strcpy( Book2.genre, "Fiction");
   strcpy( Book2.author, "JK Rowling");
// The next statements are to print the details of the first and second
variables in the structure
   cout << "Book 1 id: " << Book1.book_id <<endl;
   cout << "Book 1 title: " << Book1.book_title <<endl;
   cout << "Book 1 genre: " << Book1.genre <<endl;
   cout << "Book 1 author: " << Book1.author <<endl;
   cout << "Book 2 id: " << Book2.book_id <<endl;
   cout << "Book 2 title: " << Book2.book_title <<endl;
   cout << "Book 2 genre: " << Book2.genre <<endl;
   cout << "Book 2 author: " << Book2.author <<endl;
   return 0;
}
```

The code above will give you the following output:

```
Book 1 id: 120000
Book 1 title: Harry Potter and the Philosopher’s Stone
Book 1 genre: Fiction
Book 1 author: JK Rowling
Book 2 id: 130000
Book 2 title: Harry Potter and the Chamber of Secrets
Book 2 genre: Fiction
Book 2 author: JK Rowling
```

Using Structures as Arguments

You can use structures as arguments in a function similar to how you pass a pointer or variable as part of the function. You need to access the variables in the structure in the same way as we did in the example above.

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books book );
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
};
int main() {
  struct Books Book1; // This is where you declare the variable Book1
in the Book structure
   struct Books Book2; // This is where you declare the variable Book2
in the Book structure
// Let us now look at how you can specify the details of the first
variable
   Book1.book_id = 120000;
   strcpy( Book1.book_title, "Harry Potter and the Philosopher's
Stone");
   strcpy( Book1.genre, "Fiction");
   strcpy( Book1.author, "JK Rowling");
// Let us now look at how you can specify the details of the second
variable
   Book2.book_id = 130000;
   strcpy( Book2.book_title, "Harry Potter and the Chamber of
Secrets");
   strcpy( Book2.genre, "Fiction");
   strcpy( Book2.author, "JK Rowling");
// The next statements are to print the details of the first and second
variables in the structure
   printBook( Book1 );
```

```
    printBook( Book2 );
    return 0;
}
void printBook(struct Books book ) {
    cout << "Book id: " << book.book_id <<endl;
    cout << "Book title: " << book.book_title <<endl;
    cout << "Book genre: " << book.genre <<endl;
    cout << "Book author: " << book.author<<endl;
}
```

When you compile the code written above, you receive the following output:

```
Book 1 id: 120000
Book 1 title: Harry Potter and the Philosopher’s Stone
Book 1 genre: Fiction
Book 1 author: JK Rowling
Book 2 id: 130000
Book 2 title: Harry Potter and the Chamber of Secrets
Book 2 genre: Fiction
Book 2 author: JK Rowling
```

Using Pointers

You can also refer to structures using pointers, and you can use a pointer similar to how you would define a pointer for regular variables.

```
struct Books *struct_pointer;
```

When you use the above statement, you can use the pointer variable defined to store the address of the variables in the structure.

```
struct_pointer = &Book1;
```

You can also use a pointer to access one or members of the structure. To do this, you need to use the -> operator:

```
struct_pointer->title;
```

Let us rewrite the example above to indicate a member or the entire structure using a pointer.

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books *book );
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
};
int main() {
  struct Books Book1; // This is where you declare the variable Book1
in the Book structure
   struct Books Book2; // This is where you declare the variable Book2
in the Book structure
// Let us now look at how you can specify the details of the first
variable
   Book1.book_id = 120000;
   strcpy( Book1.book_title, "Harry Potter and the Philosopher’s
Stone");
   strcpy( Book1.genre, "Fiction");
   strcpy( Book1.author, "JK Rowling");
// Let us now look at how you can specify the details of the second
variable
   Book2.book_id = 130000;
   strcpy( Book2.book_title, "Harry Potter and the Chamber of
Secrets");
   strcpy( Book2.genre, "Fiction");
   strcpy( Book2.author, "JK Rowling");
// The next statements are to print the details of the first and second
variables in the structure
   printBook( Book1 );
   printBook( Book2 );
   return 0;
}
```

// We will now use a function to accept a structure pointer as its parameter.

```
void printBook( struct Books *book ) {
   cout << "Book id: " << book->book_id <<endl;
   cout << "Book title: " << book->book_title <<endl;
   cout << "Book genre: " << book->genre<<endl;
   cout << "Book author: " << book->author <<endl;
}
```

When you write the above code, you obtain the following output:

```
Book id: 120000
Book title: Harry Potter and the Philosopher’s Stone
Book genre: Fiction
Book author: JK Rowling
Book id: 130000
Book title: Harry Potter and the Chamber of Secrets
Book genre: Fiction
Book author: JK Rowling
```

Typedef Keyword

If the above methods are a little tricky for you, you can use an alias type to define a structure. For instance,

```
typedef struct {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
} Books;
```

This is an easier syntax to use since you can directly define all the variables in the structure without using the keyword ‘struct.’

```
Books Book1, Book2;
```

You do not have to use a typedef key only to define a structure. It can also be used to define regular variables.

```
typedef long int *pint32;
pint32 x, y, z;
```

The type long ints point to the variables x, y and z.

Chapter 4

Introduction to Object-Oriented Programming in C++

The objective of the development of C++ was to add the concept of object-oriented programming to the C programming language. The classes used in C++ programming are the variables or structures used in object-oriented programming. These classes are often termed as user-defined data types. These concepts were discussed in brief in the previous chapter.

Classes are used to define the form and type of object used in the code. This data type uses both data methods and representation to manipulate the data present in the memory into one package. The functions and data within these classes are termed as class members.

Definition of Classes

Since classes are user-defined data types, you can define how the data type should be structured. When you do this, you do not define the data to be stored in the class, but you define the name of the class. You will also define the objects used in the class and the operations you can perform on the class's objects.

Use the keyword **class** when you define the class in your code. This keyword is followed by the class name and the body of the class. You enclose these data in curly braces. You can end a class declaration with a semicolon or list of data types, declarations, and functions. The following example defines a class named triangle.

```
class Triangle {
   public:
      double length;   // This variable denotes the length of the triangle
```

```
        double height;   // This variable denotes the height of the triangle
        double breadth;  // This variable denotes the breadth of the triangle
};
```

When you use the keyword ‘public,’ it denotes that different functions in the program can access the class's attributes or members. All you need to do is call the members accurately when you want to use the values in the function. If you do not want the values of the members in the class to change, you should use the keyword ‘private.’ We will discuss this in detail in this chapter.

Defining Class Objects

You can use a class to provide the detail of how an object should be defined in the program. Every object in the class is defined in the same way you define simple variables in the code. The following statements are examples of how to declare objects in a class. We are going to declare two objects for the class Triangle.

```
//The following lines of code are used to declare the objects Triangle1
and Triangle2 in the class Triangle
Triangle Triangle1;
Triangle Triangle2;
```

Now, each of these objects will have the same members (length, breadth, and height), which we declared while defining the class Triangle.

How to Access the Class Members

If the members in the class are public members, you can access them anywhere in the code using the access operator (.).

```
#include <iostream>
using namespace std;
class Triangle {
   public:
        double length;   // This variable denotes the length of the triangle
        double height;   // This variable denotes the height of the triangle
        double breadth;  // This variable denotes the breadth of the triangle
};
```

```
int main() {
//The following lines of code are used to declare the objects Triangle1
and Triangle2 in the class Triangle
Triangle Triangle1;
Triangle Triangle2;
//The objective of the code is to calculate the area of the triangle. We
will now initialize a variable ‘area’ with the data type double and
assign it the value 0.0
double area = 0.0
   // We will now specify the parameter values for each of the class
members
   Triangle1.height = 5.0;
   Triangle1.length = 6.0;
   Triangle1.breadth = 7.0;
   Triangle2.height = 10.0;
   Triangle2.length = 12.0;
   Triangle2.breadth = 13.0;
      // We now define the function to us to calculate the area of the
triangles.
   volume = Triangle1.height * Triangle1.length * Triangle1.breadth;
   cout << "Area of the first triangle: " << volume <<endl;
   volume = Triangle2.height * Triangle2.length * Triangle2.breadth;
   cout << "Area of the second triangle: " << volume <<endl;
   return 0;
}
```

When you execute the above code, you receive the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
```

Note that you cannot access protected and private class members using the access operator (.). We will discuss how you can access protected and private class members in the code.

Classes and Objects

You now have a brief idea of what classes and objects in C++ are and how you can access the class members and objects. We will look at other aspects of object-oriented programming in further detail later in the book. Before we move onto the next chapter, let us look at some points you need to keep in mind when you work on object-oriented programming.

S. No.	Concept	Description
1	Class members and functions	You can define member functions in classes similar to the way you define member data types or variables.
2	Class access modifiers	The keywords public, private, and protected are termed as class access modifiers. If you have not defined the access modifier for the class members, the compiler takes the value as 'private.'
3	Constructors and destructors	Class constructors are special functions in C++, and these can only be used within classes, especially when you create a new class object. A destructor is another function created when you delete an object from the class.
4	Copy constructor	This function creates another object in the class by initializing and declaring the object using another object in the same class.
5	Friend functions	If defined as such in the code, Friend functions can access protected and private members present in the class.
6	Inline functions	An inline function is one that instructs the compiler to expand the entire code in the class using the details of the function without

		using a call to access the function.
7	This pointer	Objects in classes are assigned pointers, and every object has only one pointer assigned to it. This pointer only points to the memory location of the object.
8	Pointer to classes	Any pointer in a class works the same as a pointer to a structure does. It is important to note that a class is only a structure with different members, objects, and functions.
9	Static class members	Both function members and data members defined in a class can always be static class members.

Chapter 5

Differences Between Classes and Structures

We have looked at data structures and classes in detail in the last two chapters. Let us now understand the difference between a data structure and classes.

Structures and classes have similar characteristics, but there are some important differences to bear in mind. One of the most important differences is one surrounding security. Data structures are not secure, and you cannot hide any variables or members in the structure from the user. This means you cannot use data abstraction to hide any implementation details of the data, variables, and members in the structure. On the other hand, a class is secure since you can use specific keywords, such as protected and private, to hide the implementation details of the members. The following are some ways to understand this difference: The data and function members in a class are created as private members by default. Any variable or data in a structure is the default. Consider the following examples. The first program gives you a compilation error while the second one compiles accurately.

Program 1

```
// Program 1
#include <stdio.h>
  class Test {
    int x; // x is private
};
int main()
{
  Test t;
  t.x = 20; // compiler error because x is private
  getchar();
  return 0;
}
```

Program 2

```
// Program 2
#include <stdio.h>
struct Test {
    int x; // x is public
};
int main()
{
  Test t;
  t.x = 20; // works fine because x is public
  getchar();
  return 0;
}
```

When you choose to derive structures from another structure or class, the base structure's access specifier or class is public. When you derive a class from a structure or class, the default access modifier is specified as private by the compiler.

Program 3

```
// Program 3
#include <stdio.h>
class Base {
public:
    int x;
};


class Derived : Base { }; // is equivalent to class Derived : private Base
{}
int main()
{
  Derived d;
  d.x = 20; // compiler error because inheritance is private
  getchar();
  return 0;
}
```

Program 4

```
// Program 4
#include <stdio.h>
  class Base {
public:
    int x;
};
  struct Derived : Base { }; // is equivalent to struct Derived : public
Base {}
  int main()
{
  Derived d;
  d.x = 20; // works fine because inheritance is public
  getchar();
  return 0;
}
```

Chapter 6

Encapsulation in C++

Every C++ program has two elements:

- **Code or program statements**: This part of the program has many statements or actions that the compiler should perform. It holds different statements, such as methods, functions, calls to functions, etc.
- **Program data**: This section of the program is the information relevant to the program. This information in the program gets affected by different program functions and methods.

Encapsulation is another method of object-oriented programming that binds various functions and data together. These functions manipulate the different variables and data stored in the program. Encapsulation also ensures that the data is safe from external factors and interference. This concept is directly related to the concept of data hiding. Data encapsulation is the concept of combining the data and functions using the data.

C++ allows you to encapsulate the data and hide it from external factors by creating user-defined data types, known as classes. We have looked at the different access modifiers in the previous chapters. As mentioned earlier, every item in a class is labeled private by default. For instance,

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
```

```
        double breadth;     // Breadth of a box
        double height;      // Height of a box
    };
```

In the above code, the variables breadth, length, and height are termed as private variables. It also means these variables can only be accessed by members in the same class. They cannot be accessed by any other function or section of the program. This is one of the easiest ways to encapsulate data. If you want to make any section of the code public, or accessible to various sections of the code, you need to declare that these variables are public. Any variable you declare after this keyword is accessible to every section of your code. If you allow one class to access the data and functions in another class, you reduce the encapsulation in the code. The objective is to ensure that the elements in the class are private.

Example

When you write code using both public and private data members and functions, you are using both data abstractions and encapsulation. Let us look at the following example:

```
#include <iostream>
using namespace std;
class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }
      // interface to outside world
      void addNum(int number) {
         total += number;
      }
      // interface to outside world
      int getTotal() {
         return total;
      };
   private:
      // hidden data from outside world
```

```
        int total;
    };
    int main() {
       Adder a;
       a.addNum(10);
       a.addNum(20);
       a.addNum(30);
       cout << "Total " << a.getTotal() <<endl;
       return 0;
    }
```

When you run the above code, you receive the following output:

```
Total 60
```

Chapter 7

Understanding Inheritance

Inheritance is an important concept in object-oriented programming. Using this characteristic, you can define a new class in terms of an existing class in the code. This makes it easy for you to create, maintain, and update any application. Inheritance also allows you to reuse code and its functionality, thereby reducing implementation time when you develop new applications. When you create a class, you do not have to create or write a new class with new function members and data members. As a programmer, you can choose to let a new class inherit members present in an existing class. This class, known as the base class, is used by the derived class.

The objective of inheritance is to create a relationship between the base and derived classes. Consider the following statements:

1. Mammals are animals.
2. Dogs are mammals.

What do you infer from these statements? Dogs are animals. You can develop such relationships in different parts of the code.

Introduction to Base and Derived Classes

You can derive classes from one or more classes in the existing code. This means a class can inherit data, variables, class members, and function members from numerous base classes. You need to use a class derivation list to specify the child or derived class's base classes. The class derivation list has the following syntax:

class derived-class: access-specifier base-class

In the above syntax, the access specifier is the access specifier modifiers we discussed in the previous chapter – private, public, or protected. The base class is the name of an existing class in the code. If you do not use an access-specifier, then the compiler chooses the private access modifier as the default.

Consider the following example where we are using a Shape class to derive the class Rectangle.

```
#include <iostream>
using namespace std;
// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
     }
   protected:
      int width;
      int height;
};
// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
int main(void) {
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
}
```

When you run the above code, you obtain the following output:

```
Total area: 35
```

Inheritance and Access

When you use a base class to create a new class, the derived class can access every public and protected member in the base class. If you do not want a derived class to access a specific member in the base class, you should declare those variables should be declared as private members. The following table lists the different forms of access modifiers used in base classes and who can access those members.

Access Modifier	**Public**	**Protected**	**Private**
Same class	Yes	Yes	Yes
Derived class	Yes	Yes	No
Outside classes	Yes	No	No

From above, you can see that a derived class can inherit the different members in the base class as long as they are defined as public, except for the following:

1. Friend functions present in the base class
2. Overloaded operators present in the base classes
3. Destructors, constructors, and copy constructors defined in the base class

Inheritance Types

The access modifiers public, private, and protected can be used to determine what members or characteristics the derived class can derive from the base class. Most programmers do not use the access modifiers protected and private when they create derived classes. They prefer to use the public access modifier since this makes it easier for the derived class to obtain the base class's members and characteristics. You need to apply the following rules when you use inheritance in your code:

Public Inheritance

If you use a public base class to create a derived class, the members in the base class become the members of the derived class. These members will still be public members of the derived class. The derived class also obtains the protected members in the base class, and they remain protected members even in the derived class. If the base class has any private members, the derived class cannot access those members. Having said that, you can use calls to functions to access the private members through protected and public members in the base class.

Protected Inheritance

When you create a derived class from a protected base class, the protected and public members in the base class are accessible to the derived class. They become the protected members of the derived class.

Private Inheritance

When you derive members from a private base class, every member of the base class becomes a private member in the derived class.

Multiple Inheritance

A class in your code can inherit members from one or more classes in the code. You can use the following syntax for the same:

class derived-class: access baseA, access baseB...

The word access in the above syntax is the access modifier (private, public, or protected), and you need to use this keyword against any base class you create in the code. You can separate the base classes in the code using a comma. Look at the following example to understand the same:

```
#include <iostream>
using namespace std;
// Base class Shape
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
```

```
        void setHeight(int h) {
            height = h;
        }
    protected:
        int width;
        int height;
};
// Base class PaintCost
class PaintCost {
    public:
        int getCost(int area) {
            return area * 70;
        }
};
// Derived class
class Rectangle: public Shape, public PaintCost {
    public:
        int getArea() {
            return (width * height);
        }
};
int main(void) {
    Rectangle Rect;
    int area;
    Rect.setWidth(5);
    Rect.setHeight(7);
    area = Rect.getArea();
    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;
    // Print the total cost of painting
    cout << "Total paint cost: $" << Rect.getCost(area) << endl;
    return 0;
}
```

When you run the above code, you receive the following output:

```
Total area: 35
Total paint cost: $2450
```

Chapter 8

Overloading in C++

C++ also allows you to use function overloading and operator overloading in the code. You can specify more than one operator in the scope of the same function or give different definitions to a function name. When you call an overloaded operator or function in the code, you are giving the compiler the liberty to choose the function name definition or operator in the current section of the code. The compiler does this based on the parameters used in the function and its definition. This process where the compiler chooses the appropriate overloaded operator or function is called overload resolution.

Introduction to Function Overloading

You can always define a function name in different ways in the same code or scope. If you want to use function overloading in your code, you need to define the function differently in the code. You can do this by using different parameters or arguments in the function and types. It is important to note that you cannot use overload function declarations only by adding a different return type. The following is an example where we are using the print() function to look at different data types in the code:

```
#include <iostream>
using namespace std;
class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
```

```
        void print(char* c) {
          cout << "Printing character: " << c << endl;
        }
};
int main(void) {
   printData pd;
   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");
   return 0;
}
```

When you compile the above code, you receive the following output:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Introduction to Operator Overloading

C++ allows you to overload or redefine most operators built-in in the code. Therefore, as a programmer, you can use different operators even if you have user-defined types. An overloaded operator is a type of function with a special name – the keyword ‘operator.’ You need to define the operator's symbol after the keyword to define the overloaded operator. When you define an overloaded operator, you also need to define the parameter list and return type. Consider the following example:

```
Triangle operator+(const Triangle&);
```

In the above example, we are declaring the addition operator you can use to add two triangle objects. It then returns the output for the Triangle object. You can define an overloaded operator as an ordinary non-member function. Alternatively, you can define the operator using a class member function. In case you want to define the above function using a non-member function in the class, you need to pass two arguments in the following manner:

```
Triangle operator+(const Triangle&, const Triangle&);
```

The following is an example of how you can use operator overloading using member functions. We pass an object as an argument in a function, and the function accesses the properties of the argument using the object. The object will then call the operator, and this object can be accessed using the operator.

```
#include <iostream>
using namespace std;
class Triangle {
   public:
      double getArea(void) {
         return 0.5*length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }
      // In this section, we are looking at the overload addition operator
to add two triangle objects.
      Triangle operator+(const Triangle& b) {
         Triangle;
         triangle.length = this->length + t.length;
         triangle.breadth = this->breadth + t.breadth;
         triangle.height = this->height + t.height;
         return triangle;
      }
         private:
      double length;      // Length of a triangle
      double breadth;     // Breadth of a triangle
      double height;      // Height of a triangle
};
// Main function for the program
```

```
int main() {
//The next three statements are used to declare three triangle objects
   Triangle Triangle1;
   Triangle Triangle2;
   Triangle Triangle3;
   double area = 0.0;     // We are declaring the variable area which will
be used to store the area of the three objects.
    // We will now specify the values for the variables defined for each
of the objects
   Triangle1.setLength(6.0);
   Triangle1.setBreadth(7.0);
   Triangle1.setHeight(5.0);


   Triangle2.setLength(12.0);
   Triangle2.setBreadth(13.0);
   Triangle2.setHeight(10.0);
    // Let us now calculate the area of the two objects
  area = Triangle1.getArea();
   cout << "Area of the first triangle: " << area <<endl;
   area = Triangle2.getArea();
   cout << "Area of the second triangle: " << area <<endl;
   // We will now add the area of the first two triangles and store it in a
third triangle
   Triangle3 = Triangle1 + Triangle2;
   // The area of the third triangle is calculated as follows
   Area = Triangle3.getArea();
   cout << "Area of the third triangle: " << area <<endl;
   return 0;
}
```

When you run the above code, you will receive the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
Area of the third triangle: 2700
```

Non-Overloadable or Overloadable Operators

The following are some operators that you can use for operator overloading.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

The following are some operators you cannot overload in your code:

::	.*	.	?:

Example of Operator Overloading

The following are some examples of operator overloading to help you understand the concept of operator overloading:

Unary Operator Overloading

A unary operator, as the name suggests, can only operate on one operand in the code. The following are some example of unary operators:

- The decrement and increment operators, -- and ++ respectively
- The not (!) logical operator
- The minus (-) unary operator

You can use these unary operators to work on specific objects. The operator always appears on the left of the object, as in ++obj, --obj, !obj, and -obj. You can also use the operators on the right side of the object if needed. The following is an example where we overload a minus operator.

```
#include <iostream>
using namespace std;
class Distance {
   private:
      int feet;            // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};
int main() {
   Distance D1(11, 10), D2(-5, 11);
   -D1;                  // apply negation
   D1.displayDistance();    // display D1
   -D2;                  // apply negation
   D2.displayDistance();    // display D2
   return 0;
}
```

When you run the above code, you will obtain the following output:

```
F: -11 I:-10
F: 5 I:-11
```

You can use the above example if you want to overload any of the unary operators mentioned above.

Binary Operator Overloading

A binary operator takes two variables or arguments. The following are some examples of binary operators:

- Addition (+)
- Subtraction (-)
- Division (/)

The following is an example of how you can use the different operators mentioned above.

```
#include <iostream>
using namespace std;
class Triangle {
//The following are the data members or variables of the class triangle
   double length;
   double breadth;
   double height;
      public:
    double getArea(void) {
      return 0.5 * length * breadth * height;
   }
      void setLength( double len ) {
      length = len;
   }
    void setBreadth( double bre ) {
      breadth = bre;
   }
    void setHeight( double hei ) {
      height = hei;
   }
    // We are now going to use the addition operator to perform an
overload on the existing values in the classes
   Triangle operator+(const Triangle& t) {
      Triangle triangle;
      triangle.length = this->length + t.length;
      triangle.breadth = this->breadth + t.breadth;
```

```
        triangle.height = this->height + t.height;
        return triangle;
    }
};
// Main function for the program
int main() {
//Declare the three objects Triangle1, Triangle2 and Triangle3
    Triangle Triangle1;
    Triangle Triangle2;
    Triangle Triangle3;
//Declaring the variable area to store the area of the triangle
    double area = 0.0;
     // We will now specify the 1 specification
    Triangle1.setLength(6.0);
    Triangle1.setBreadth(7.0);
    Triangle1.setHeight(5.0);
     Triangle2.setLength(12.0);
    Triangle2.setBreadth(13.0);
    Triangle2.setHeight(10.0);
     area = Triangle1.getVolume();
    cout << "Area of the first triangle: " << area <<endl;
     area = Triangle2.getVolume();
    cout << "Area of the second triangle: " << area <<endl;
     // We will now calculate the area of the third triangle using the
binary operator
    Triangle3 = Triangle1 + Triangle2;
     area = Triangle3.getArea();
    cout << "Area of the third triangle: " << area <<endl;
     return 0;
}
```

When you run the above code, you obtain the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
Area of the third triangle: 2700
```

Relational Operator Overloading

C++ supports different relational operators, such as:

- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Equal to (==)

You can overload the above operators in C++ and use them to compare the values of one object in a class to another. The following example shows you how you can use the less-than operator and overload it. You can similarly overload the other relational operators:

```
#include <iostream>
using namespace std;
class Distance {
   private:
      int feet;            // 0 to infinite
      int inches;          // 0 to 12
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }
         return false;
      }
};
int main() {
   Distance D1(11, 10), D2(5, 11);
    if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }
     return 0;
}
```

When you run the above code, you receive the following output:

```
D2 is less than D1
```

Input and Output Operator Overloading

You can use the stream extraction operator (>>) and stream insertion operator (<<) to use built-in data types as inputs and outputs. C++ allows you to overload these operators and perform different input and output operations on the different classes and user-defined objects. That said, you need to convert the operator overloading function into a friend function for the class since you call it without having to create the object. The following is an example of how you can use the insertion and extraction operator and overload it.

```
#include <iostream>
using namespace std;
class Distance {
    private:
        int feet;            // 0 to infinite
        int inches;           // 0 to 12
    public:
        // required constructors
        Distance() {
            feet = 0;
            inches = 0;
        }
        Distance(int f, int i) {
            feet = f;
            inches = i;
        }
        friend ostream &operator<<( ostream &output, const Distance
&D ) {
            output << "F : " << D.feet << " I : " << D.inches;
            return output;
        }
        friend istream &operator>>( istream  &input, Distance &D ) {
            input >> D.feet >> D.inches;
            return input;
        }
};
int main() {
    Distance D1(11, 10), D2(5, 11), D3;
    cout << "Enter the value of object : " << endl;
    cin >> D3;
    cout << "First Distance : " << D1 << endl;
    cout << "Second Distance :" << D2 << endl;
    cout << "Third Distance :" << D3 << endl;
    return 0;
}
```

When you compile and execute the above code, you obtain the following result:

```
        $./a.out
Enter the value of object :
        70
        10
        First Distance : F : 11 I : 10
        Second Distance :F : 5 I : 11
        Third Distance :F : 70 I : 10
```

Assignment Operator Overloading

C++ allows you to overload the assignment operator in the same way you overload other operators in C++. You can use the overloaded object to create an object in the same way you use a copy constructor for the same. The following is an example of how you can overload an assignment operator in C++.

```
#include <iostream>
using namespace std;
class Distance {
   private:
      int feet;          // 0 to infinite
      int inches;          // 0 to 12
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }
      // method to display distance
```

```
        void displayDistance() {
            cout << "F: " << feet <<  " I:" <<  inches << endl;
        }
};
int main() {
    Distance D1(11, 10), D2(5, 11);
    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance();
    // use assignment operator
    D1 = D2;
    cout << "First Distance :";
    D1.displayDistance();
    return 0;
}
```

On running the above code, you obtain the following output:

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

Function Call Operator Overloading

You can overload the function call operator on any object in the class. When you overload the function operator, you create a new way to call the function and create a new operator function using which you can pass numerous arbitrary parameters. The following is an example of how you can overload the operator:

```
#include <iostream>
using namespace std;
class Distance {
    private:
        int feet;             // 0 to infinite
        int inches;           // 0 to 12
    public:
        // required constructors
        Distance() {
           feet = 0;
           inches = 0;
        }
        Distance(int f, int i) {
           feet = f;
           inches = i;
        }
        // overload function call
        Distance operator()(int a, int b, int c) {
           Distance D;
          // just put random calculation
           D.feet = a + c + 10;
           D.inches = b + c + 100 ;
           return D;
        }
        // method to display distance
        void displayDistance() {
           cout << "F: " << feet << " I:" << inches << endl;
        }
  };
  int main() {
     Distance D1(11, 10), D2;
     cout << "First Distance : ";
     D1.displayDistance();
     D2 = D1(10, 10, 10); // invoke operator()
     cout << "Second Distance :";
     D2.displayDistance();
     return 0;
  }
```

When you run the above code, you will obtain the following result:

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

Subscript Operator Overloading

A subscript operator is used to access different elements in an array. You can overload this operator to enhance or improve the functionality of arrays in C++. The following is an example of how you can overload the operator:

```
#include <iostream>
using namespace std;
const int SIZE = 10;
class safearay {
   private:
      int arr[SIZE];
   public:
      safearay() {
         register int i;
         for(i = 0; i < SIZE; i++) {
           arr[i] = i;
         }
      }
      int &operator[](int i) {
         if( i > SIZE ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
         }
         return arr[i];
      }
};
int main() {
   safearay A;
   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;
   return 0;
}
```

On running the above code, the output received is:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
```

```
Value of A[12] : 0
```

Chapter 9

Polymorphism in C++

Polymorphism is a concept in C++ which allows classes to have multiple forms. This only occurs when you have numerous classes in the code related through inheritance. Polymorphism in C++ means that the object determines the type of function to be executed by the compiler. When you call a function within a class, the compiler will look at the object type being used as an argument or parameter in the function before it invokes the function relevant to the object. In the example, we will look at how we can derive a base class based on two other classes.

```
#include <iostream>
using namespace std;
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }
      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// This is the main function of the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
  // The variable shape is used to store the values for the rectangle
   shape = &rec;
   // This statement is used to call the function to calculate the area of
the rectangle
   shape->area();
  // The variable shape is used to store the values for the triangle
   shape = &tri;
   // This statement is used to call the function to calculate the area of
the triangle
   shape->area();
     return 0;
}
```

When the above code is run, the following is the output you will receive on your screen:

```
Parent class area :
Parent class area :
```

This is not the output we want. If you are unable to identify the reason for this incorrect output, let me tell you what change needs to be made to the code. Look at how the function area() is being called. We have defined this function in the base class, and the compiler uses this version. This method of linking functions is termed as static linkage or resolution. The call of this function should be fixed before you execute the program. This process is termed early binding. The compiler will set this function when it debugs the program. Let us now make a slight change to the above program. We will declare the area function within the Shape class itself. We will also use the virtual keyword. The updated code is as follows:

```
class Shape {
   protected:
      int width, height;
      public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

When you make this modification to the code, you will receive the following output:

```
Rectangle class area
Triangle class area
```

The compiler will now look at the pointer and the contents of the variable it is pointing to. It will no longer look at the data type. The compiler calls the area function since the rec and tri class objects are not stored in shape but stored in *shape. Every child class in the above code has its own implementation of the area function. This is how programmers use the polymorphism concept in C++. The above code has different classes, and each class has the same function. The functions also take the same parameters, but the implementation of the function is different in each case.

Understanding Virtual Functions

When you define a function using the virtual keyword, it becomes a virtual function. C++ allows you to define a virtual function in the base class and a different version of the virtual function in the derived class. This only signals to the compiler that you do not want any link to exist between the functions. The only thing you need to be aware of is the position of the function you want to call in the code. To do this, you need to ensure the function selected by the compiler is based on the object you want to use it on. This type of connection or link is termed as late binding or dynamic linkage.

Pure Virtual Functions

You can include virtual functions in the base class and use that in a derived class. It is important to note that these functions can be redefined in the derived classes, so the functions suit the objects present in the derived class. There is, however, no meaningful definition you can give the function in the class. The following example shows you how you can change a virtual function in the base class.

```
class Shape {
   protected:
      int width, height;
   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

We see that the virtual function area() has been assigned the value zero in the above code. This indicates to the compiler that the function does not have a body. This is an example of a pure virtual function.

Chapter 10

Abstraction in C++

Data abstraction is the process of hiding details or information in the code from other functions or classes in the code. The objective of data abstraction is to only present the details relevant to the other classes without sharing the actual details present in the class. Abstraction is a design and programming technique that relies only on two aspects – interfaces and implementation.

Consider the following example. When you use a television, you can switch it on and off, switch between channels, add speakers and other external devices, such as DVDs and VCRs or even increase or decrease the volume. You can do this using a remote, but you do not know what happens inside the device for you to be able to do this. You do not know how the signals pass through the cables or air or how the television interprets those signals before it displays them on the screen. Therefore, we can definitively state that a television will separate the interface's internal functionalities or implementation. You can use a remote to play with the external interface without learning anything about the internal components.

C++ gives every class you define some level of abstraction. When you define a class, you can determine the different class methods that the other classes in the code can use. This allows those classes to manipulate the data members and function members in the class without knowing how the base class works. For instance, you can use the sort function anywhere in your code without knowing what the function does or the algorithm it uses. Bear in mind that the functionality and algorithm used in the sort function will vary from one release to the next. If the interface being used still stays the same, any call made to the function will work in the code.

You can define abstract data types (ADTs) in the classes defined in the code. You can also use the cout object from the ostream class to move data into standard output files.

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello C++" <<endl;
   return 0;
}
```

From the above code, it is clear you do not have to worry about how the function cout works. You know it is used to display the output on the screen. It is only important for you to know the public interface used since the implementation of the cout function or keyword can change.

Benefits

There are two advantages of using abstraction in your code:

- The internal members of the class are always protected from any errors which users are bound to make. This prevents the corruption of objects in the classes.
- The implementation of classes and functions can evolve or change over time in response to the needs or requirements that constantly change. The developers may also make fixes or implementation changes in case of any bug report.

When you define data members or function members in the class, especially in private sections in the class, you can make any changes to the data as the author of the class. If the implementation changes, you only need to examine the code written in the class to see how the implementation affects the classes. If the data used in the classes is public, then a function related to the data or function members in the class may break in case of an implementation change.

How to Enforce Abstraction

In C++, you can use different access labels to define the interfaces in the class where you want to protect the data. Any class you define can contain zero, one, or more labels.

- When you define a class member using the public label, you allow that member to be accessible to any function in the program. Public members determine how the members in the class are viewed.
- If you define members using the private or protected labels, these values will not be accessible to any function or class in the code. The implementation of the members will always remain hidden.

C++ does not restrict the number of times you use an access label in your code. Every access label used in the code specifies the level of access every member in the firm has. The specific access level will always remain in effect until the compiler comes across another access level in the code.

Example

When you write a code with private or public class members, you are using the concept of data abstraction.

```
#include <iostream>
using namespace std;
class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }
      // interface to outside world
      void addNum(int number) {
         total += number;
      }
      // interface to outside world
      int getTotal() {
         return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main() {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

When you run the above code, you will obtain the following output:

```
Total 60
```

In the above code, we add two numbers and return the sum of those numbers. We have two public members in the code:

1. addNum
2. getTotal

The compiler uses these members as the interface to the outside world. A user only needs to know how to work with the class.

The private member in the code is total, and this is a value the user does not know about. The class, however, needs this variable for it to function.

Why Use Abstraction?

Through abstraction, you can separate the code into two parts – implementation and interface. When you design any code or program component, make sure the interface and implementation are independent. This is the only way you can ensure that any change made to the implementation would not affect the interface. This helps you control the functioning of any program to ensure there is no impact on the component and application because changes are made to the implementation.

Chapter 11

Abstract Classes or Interfaces

You may need to describe or use classes in your programs or code without committing or linking the class to a specific type of implementation. You can do this using an interface. You can implement interfaces in C++ using a concept termed as an abstract class. Do not confuse an abstract class with the concept of data abstraction. The latter is a concept used in object-oriented programming wherein the code's implementation details are kept away or apart from the interface and data used in the code.

You can make any class abstract by declaring a pure virtual function. We discussed this in an earlier chapter. a pure virtual function is one where the value of the function is equated to zero. For example:

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Most programmers use an abstract class as a base class using which they create derived classes. An abstract class is one type of class using which you cannot create, declare, assign, or initialize an object. These classes only serve the purpose of an interface. If you try to create an object and instantiate it in an abstract class, it only leads to compilation errors. Therefore, if you want to instantiate a subclass or object in an abstract class, you need to implement numerous virtual functions. Virtual functions can support any interface created or declared within an abstract class. If you do not override a pure virtual function in any derived class and declare or instantiate an object in the class, it will lead to an error. These mistakes are very small but hard to detect in the code. You can also create a concrete class to instantiate and declare objects.

Example

The following is an example of how you can use abstract classes to implement functions. In the example below, we will look at how you can implement the function getArea().

```
#include <iostream>
using namespace std;
// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }
   protected:
      int width;
      int height;
};
// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};
int main(void) {
   Rectangle Rect;
   Triangle  Tri;
   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;
   Tri.setWidth(5);
   Tri.setHeight(7);
  // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;
  return 0;
}
```

When you run the above code, you receive the following output:

```
Total Rectangle area: 35
Total Triangle area: 17
```

In the above example, we see how you can use an abstract class to define an interface using the function getArea(). We also saw how you could implement this function in two other classes in the code. Each of these classes, however, uses a different algorithm to calculate the area of the shape.

When you develop an application using object-orientation, you may need to use an abstract class to provide a standard and common interface. This interface will be appropriate for any external class, application, or function you may want to use. Through inheritance, the derived classes obtain the necessary methods and data from the abstract base class. The functions, classified as public in the abstract class, should be pure virtual functions. These pure functions can only be used in the derived classes, which correspond to the specific functions and methods used in the application.

This makes it easier for you, as a programmer, to add new objects and applications to the existing code even after you have defined the system.

Chapter 12

Constructors in C++

As mentioned earlier, a constructor is a function created in a class when you create an object. This function is used to initialize the object in a class. A constructor is termed as a member function in every class. Before we look at the details of what constructors are, let us understand the difference between constructors and member functions. The following are some differences between constructors and member functions in a class:

- A constructor, unlike a member function, will have the same name as the class
- There is no return type for a constructor
- When you create an object in a class, the compiler automatically creates a constructor. You need to create a member function separately if you want to perform any operations.
- You need not define any constructor in the code since the compiler automatically generates one with no body or parameters

Let us understand what a constructor is better using an example. Let us assume you went to the store to purchase a pen. Do you consider all the options available when you choose to buy a pen? The first thing you do is think about the store you want to go to. Once you reach the store, you ask the shopkeeper to give you a pen. When you ask for just a pen, it indicates you have not thought about the brand you want to use or which color you prefer. The shopkeeper will hand you a pen or give you a pen that people have bought frequently. This is exactly what a default constructor in your class is.

The other option you have is to go to the store and let the shopkeeper know you want a blue color pen sold by ABC brand. When you mention this to him, he will hand the exact product to you. The shopkeeper knows what you want because you gave him the parameters. This is an example of a parameterized constructor.

The last option is to take a pen you have at home and show the shopkeeper a physical copy of the pen and ask for the same thing. The shopkeeper will give you exactly that pen. This new pen is a copy of the pen you own. This is how a copy constructor works.

Constructor Types

The following are the types of constructors we discussed previously.

Default Constructors

A default constructor is one that does not use any parameters or arguments. There is also no function or operation defined within this constructor.

Consider the following example:

```
// This program is an example of a default constructor
#include <iostream>
using namespace std;
class construct
{
public:
    int a, b;
    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
```

```
              << "b: " << c.b;
        return 1;
    }
```

On compiling the code, you will receive the following output:

```
    a : 10
    b : 20
```

It is important to note that the compiler always defines a constructor in the code when you create an object in the class.

Constructors with Parameters

Constructors are like functions in the sense that you can pass arguments or parameters. These parameters or arguments allow you to declare and initialize an object in the class when you create it. If you want to create a constructor that accepts parameters and arguments, you need to add them to the constructor, similar to how you would add them to functions. When you define the body of the constructor, you should use the parameters or arguments to initialize or instantiate the objects in the class.

Consider the following example:

```
// This example is used to create a constructor with parameters and
arguments
#include <iostream>
using namespace std;
class Point
{
private:
    int x, y;
public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }
     int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};
int main()
{
    // Constructor called
    Point p1(10, 15);
     // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
     return 0;
}
```

When you run the above code, you will obtain the following output:

```
p1.x = 10, p1.y = 15
```

When you use a parameterized constructor to create or declare an object, you need to pass the values you want to assign the object as an argument or parameter in the function. The compiler will not allow you to declare and assign a value to an object normally. Therefore, you need to call a constructor, and you can either do this implicitly or explicitly.

Consider the following example:

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);          // Implicit call
```

Uses of Parameterized Constructors

Parameterized constructors can be used for the following:

1. You can use this constructor to initialize different data elements in the code with different objects. Each of these objects can be assigned different values depending on when they are created.

2. You can use this function for constructor overloading. This method is similar to the process of overloading discussed above.

Copy Constructors

Copy constructors are member functions using which you can initialize an object in the classes. You can do this by using other objects in the same class. We will look at these in further detail later in the book.

When you define more than one constructor in the class with parameters, you also need to declare a default constructor without parameters. This should be declared explicitly in the class. The compiler will not create a default constructor if you have defined a constructor in the code. It is, however, not required for you to always declare a default constructor. However, this is the best practice.

```
// Example to create a copy constructor
#include "iostream"
using namespace std;
class point
{
private:
  double x, y;
public:
  // Non-default Constructor &
  // default Constructor
  point (double px, double py)
  {
    x = px, y = py;
  }
};
int main(void)
{
  // Define an array of size
  // 10 & of type point
  // This line will cause error
  point a[10];
  // Remove above line and program
  // will compile without error
```

```
        point b = point(5, 6);
    }
```

The following is the output when the code above is compiled:

```
Error: point (double px, double py): expects 2 arguments, 0 provided
```

Chapter 13

Copy Constructors in C++

We looked at the different types of constructors in the previous chapter. In this chapter, we will look at a copy constructor in further detail.

Definition

A copy constructor, like other constructors, is a member function in the class. This constructor is a copy of an existing member function or constructor in the class. You can use this function to initialize an object in the same class. Every copy constructor has the general syntax:

```
ClassName (const ClassName &old_obj);
```

The following is an example of how you can create or use copy constructors.

```
#include<iostream>
using namespace std;
class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }
    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }
    int getX()          {  return x; }
    int getY()          {  return y; }
};
int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

When you run the above code, you receive the following output:

```
    p1.x = 10, p1.y = 15
    p2.x = 10, p2.y = 15
```

When Do You Call a Copy Constructor?

The following are the instances when you can call a copy constructor.

1. When the member functions in the class return the object as a value
2. When you pass the object as a parameter or argument in a function instead of declaring it in the class
3. When you construct an object using a different object present in the same class
4. When the compiler uses the constructors and member functions to create temporary objects

However, you do not have to create a copy constructor or use it in any of these situations since C++ lets the compiler take the decision to save the memory. The compiler can choose to update and monitor the way a copy constructor is called, and objects and functions are copied.

When Should You Define a Copy Constructor?

If you do not define the copy constructor, the compiler will create a default constructor for every class. This does not necessarily have to be a copy made as per the member functions and data. The compiler then creates a copy constructor, which will work fine in general. When a compiler creates a copy constructor, it works like every other copy constructor. If you want to define a copy constructor for a certain function, make sure the object has a pointer attached to it.

If you let the compiler create a copy constructor, it will only be a shallow copy of the constructor. You can only ensure that the object or class is fully copied by creating a user-defined copy constructor. In these constructors, you can define the references and pointers of copied objects to every location.

Assignment Operators Versus Copy Constructors

From the example below, which statement do you think will call the assignment operator and which one will call the copy constructor?

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

The compiler will call the copy constructor when you create a new object in the code from an existing object. It only calls the constructor since you create a copy of the existing copy. It calls the assignment operator when you assign a value to an existing object in the code. In the example above, the first statement will call the copy constructor, and the second constructor will call upon the assignment operator.

Example Where You Use Copy Constructors

The example below shows how you can use copy constructors in C++. We will define a copy constructor in the String class.

```
#include<iostream>
#include<cstring>
using namespace std;
  class String
{
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s;  }// destructor
    String(const String&); // copy constructor
    void print() { cout << s << endl; } // Function to print string
    void change(const char *);  // Function to change
};
  String::String(const char *str)
{
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
  void String::change(const char *str)
{
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
  String::String(const String& old_str)
{
    size = old_str.size;
    s = new char[size+1];
    strcpy(s, old_str.s);
}
  int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;
      str1.print(); // what is printed ?
    str2.print();
      str2.change("GeeksforGeeks");
      str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

The output of this code is:

```
GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks
```

What Happens When You Remove a Copy Constructor From the Code?

When you remove copy constructors from the code above, you will not receive the output. The changes you make to variable str2 will reflect the value in str1, which is not the expected outcome.

```
#include<iostream>
#include<cstring>
using namespace std;
class String
{
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s;  }// destructor
    void print() { cout << s << endl; }
    void change(const char *);  // Function to change
};
String::String(const char *str)
{
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
void String::change(const char *str)
{
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;
    str1.print(); // what is printed ?
    str2.print();
    str2.change("GeeksforGeeks");
    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

The output of the above code is:

```
GeeksQuiz
GeeksQuiz
GeeksforGeeks
GeeksforGeeks
```

Can Constructors be Made Private?

You can make copy constructors private variables in the code. It is important to bear in mind that every object or member of the class can no longer be copied. It is useful to do this only when the class you define the constructor in has pointers, or the resources are allocated memory space dynamically. In these situations, you can write a copy constructor like the above example. You can also make the constructor private so that a user will receive a compiler error instead of runtime errors.

Passing Arguments in Copy Constructors

You can only pass arguments in a copy constructor in the form of a reference. When you pass an argument using a value in the copy constructor, a call to the constructor will lead to an error since the compiler only looks at the constructor and not the values of the arguments passed as parameters.

Chapter 14

Destructors in C++

Now that we have looked at what a constructor is let us learn more about destructors. A destructor is another member function that is created by the compiler when you delete any object in the existing class. The syntax used for a destructor is as follows:

~constructor-name();

Properties

The following are properties of destructors:

- A destructor function is created by the compiler when you delete an object in a class
- You cannot declare a constant or static member function
- C++ does not allow you to enter any arguments
- The destructor does not have any return type. You cannot enter void as a return type
- If you have an object in a destructor class, it does not become a member of the class
- Destructors can only be declared in the public section of any class
- You do not have access to the address of the destructor function

When Do You Call a Destructor?

The compiler calls a destructor function when the object is no longer out of scope.

1. The program ends
2. The function ends
3. An object is deleted using the delete operator
4. A block containing the local variable will end

Difference Between Destructors and Member Functions

A destructor will have the same name as the class where an object has been deleted. The only way to differentiate between a destructor and class is the presence of a tilde (~). a destructor does not take arguments or parameters and cannot return any value.

Consider the following example:

```
class String {
private:
    char* s;
    int size;
public:
    String(char*); // constructor
    ~String(); // destructor
};
String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() { delete[] s; }
```

Can You Have More Than One Destructor?

Most people wonder if you can have multiple destructors in a class. If you delete more than one object, the compiler is bound to create more than one destructor, right? This, however, is not a correct assumption. There can only be one destructor in a class since the function takes the name of the class. There will be no parameters or functions within the destructor.

When Should You Define a Destructor?

The compiler creates a destructor automatically in the code if you do not declare one. It is okay to use the default destructor if none of the objects are allocated memory space dynamically. A default destructor does not work the way it should if you have a pointer. When classes contain pointers, the memory allocated to the class object will first need to be removed before you delete the instance. This is the only way to prevent a memory leak.

Can You Define Virtual Destructors?

Experts recommend that you describe a virtual destructor. We will look at virtual destructors in detail in the following chapter.

Chapter 15

Virtual Destructors in C++

You can create a derived class where the objects use pointers to obtain the information of relevant objects from the base class. If you delete such objects using a non-virtual destructor, it will lead to runtime errors. To improve or rectify this situation, you need to define a virtual destructor in the base class. If you were to compile the code in the example below, it would result in an error in the code.

```
// This program is an example of how a class without a virtual
destructor leads to runtime errors in the code
#include<iostream>
  using namespace std;
  class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
  class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
  int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

The above code's output may be slightly different between different C++ instances depending on the compiler used. If you use the Dev-CPP compiler, you will obtain the following output:

```
Constructing base
Constructing derived
Destructing base
```

When you create a virtual destructor in the base class in the code, it indicates to the compiler that the object in the derived class, when deleted, will be removed correctly from the code. These lines of code ensure that both the derived and base class destructors are called when you run the code. For instance,

```
// This program is an example of how you can use a virtual destructor
to avoid a runtime error
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
```

```
int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

You will receive the following output when you run the above code:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

If you use a virtual function in your class, you need to add a virtual destructor to the code immediately, regardless of whether or not you use it. This is the only way you can prevent any runtime errors.

Pure Virtual Destructors

You can create a pure virtual destructor in C++ if you need to. When you define these functions in a class, you need to add a body to the destructor. This contradicts the definition of a virtual function, doesn’t it? Why does a virtual function need a body? We need to do this since the compiler does not override the destructor. The compiler calls a pure virtual destructor in the reverse order when a class is derived. This indicates that a destructor in a class is invoked first by the compiler. It is only after this that the compiler calls the destructor in the class.

If you do not define a pure virtual destructor in the code, what function or statements will the compiler call upon when it needs to destroy or delete an object? It is only when you define the right objects that the linker and compiler in the code enforce the presence of the function body. Consider the example below:

```
#include <iostream>
class Base
```

```
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};
class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed";
    }
};
  int main()
{
    Base *b=new Derived();
    delete b;
    return 0;
}
```

The linker in the code will give you the following error:

```
test.cpp:(.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):
undefined reference to `Base::~Base()'
```

If you define a pure virtual destructor in the code, the program will compile the code without any errors.

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};
Base::~Base()
{
    std::cout << "Pure virtual destructor is called";
}
class Derived : public Base
```

```
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed\n";
    }
};
int main()
{
    Base *b = new Derived();
    delete b;
    return 0;
}
```

The output of the above code is:

```
~Derived() is executed
Pure virtual destructor is called
```

Bear in mind that a class in your code becomes an abstract class if you have pure virtual destructors. Consider the program below. Write it in your C++ window and see how it runs.

```
#include <iostream>
class Test
{
public:
    virtual ~Test()=0; // Test now becomes abstract class
};
Test::~Test() { }
int main()
{
    Test p;
    Test* t1 = new Test;
    return 0;
}
```

When you run the above code in the compiler, you receive the following messages:

```
[Error] cannot declare variable ‘p’ to be of abstract type ‘Test’
[Note] because the following virtual functions are pure within ‘Test’:
[Note] virtual Test::~Test()
[Error] cannot allocate an object of abstract type ‘Test’
[Note] since type ‘Test’ has pure virtual functions
```

If you make the changes indicated in the error to the code, the program compiles without any errors.

Chapter 16

Introduction to Private Destructors

```
// This program is an example of a private destructor
#include <iostream>
using namespace std;
  class Test {
private:
    ~Test() {}
};
int main()
{
}
```

If you run the above code, you will see it compile with no errors. Therefore, you can say that there is no compiler error in the code, which indicates that the compiler does not throw an error when it comes across a private destructor. Consider the program below:

```
// This program is used to explain how a private destructor functions
#include <iostream>
using namespace std;
class Test {
private:
    ~Test() {}
};
int main()
{
    Test t;
}
```

When you run the above code, you will receive a compile error. The compiler notes that you have declared a variable ‘t’ and it cannot delete it from the class or code since the destructor you have defined is private. What do you think happens in the following code?

```
// Code to understand private destructors
#include <iostream>
using namespace std;
  class Test {
private:
    ~Test() {}
};
int main()
{
    Test* t;
}
```

When you run the above code, you do not receive any error. There is no object constructed as part of the code, and the compiler uses the pointer. Therefore, there is no use of a destructor. Now, what about the following program?

```
// Example of a private destructor
  #include <iostream>
using namespace std;
  class Test {
private:
    ~Test() {}
};
int main()
{
    Test* t = new Test;
}
```

When you run the above code, you will not receive any error in the code. Why do you think this is the case? The above code uses dynamic memory allocation to store the variables. Therefore, it is your duty to delete the object stored in the dynamic memory. It is for this reason why the compiler does not care.

In case you create a destructor and label it as a private member function, you can also create another instance in the class using the malloc() function (memory allocation). Consider the following example:

```
// Example of a private destructor
  #include <bits/stdc++.h>
using namespace std;
  class Test {
public:
    Test() // Constructor
    {
        cout << "Constructor called\n";
    }
  private:
    ~Test() // Private Destructor
    {
        cout << "Destructor called\n";
    }
};
  int main()
{
    Test* t = (Test*)malloc(sizeof(Test));
    return 0;
}
```

You do not receive any output when you run the code. The following code fails to compile accurately.

```
// Example of a private destructor
#include <iostream>
using namespace std;
class Test {
private:
    ~Test() {}
};
int main()
{
    Test* t = new Test;
    delete t;
}
```

When you create a class with a private destructor, a dynamic object is created for those classes. The following example is one where you can create a class using a private destructor. You can also create a friend function in the class. This function is only used to delete objects.

```
// Example of a private destructor
#include <iostream>
  // a class with private destructor
class Test {
private:
    ~Test() {}
    friend void destructTest(Test*);
};
  // Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
    delete ptr;
}
  int main()
{
    // create an object
    Test* ptr = new Test;
      // destruct the object
    destructTest(ptr);
      return 0;
}
```

When you want to control or monitor the destruction or deletion of objects in a class, you need to make the destructor private. If you use dynamically created objects in your classes, you can delete the object by passing pointers in the function as arguments or parameters. These functions delete the objects in the code. It is important to note that a reference to an object after the function is called will lead to a dangler.

Chapter 17

Exception Handling in C++

C++ is very different from C in terms of exception handling. An exception is any abnormal condition or a runtime error in the code. These are anomalies the compiler encounters when it executes the lines of code written. The following are the types of exceptions you may encounter when the compiler runs the program:

1. Synchronous
2. Asynchronous

An asynchronous error is beyond the compiler's control and the code written in the application. If you want to prevent the occurrence of an asynchronous error, you can use the following keywords:

1. *Try*: If you add this keyword at the beginning of a block of code, you can indicate to the compiler that this block of code may throw an error or exception
2. *Catch*: This keyword depicts that block of code that should be executed when the compiler throws a specific error
3. *Throw*: This keyword throws an exception or lists the different exceptions the block of code may throw. The compiler can ignore these errors and avoid handling them if it chooses to.

Importance of Exception Handling

The following are the reasons why you need to use or include error handling code or keywords in your program:

Separation of Normal Code From Error Handling Code

If you use traditional methods to write error handling codes, you can include if-else functions and other conditional statements to handle any errors. This only leads to confusion since the error handling code gets mixed with the normal flow, which makes it difficult for you to read the code. It also becomes difficult to maintain the code. If you use try-catch blocks of code in your program, you can separate the normal code from the error handling code.

Methods and Functions Can Choose How to Handle Exceptions

Functions and methods have different operations within the body, which may throw some exceptions, but these functions and methods can choose to handle these exceptions differently. If the function is called elsewhere in the program, the compiler can choose how to handle the other exceptions in the code. If the function caller does not care about the exceptions or misses them, then the caller of the caller will need to handle the error in the code.

When you write code in C++, you can write the throw keyword against the statements where you think there will be an error. The caller of the function with this keyword will need to identify a way to handle the exceptions. It can either catch the error or specify it again to the compiler.

Grouping Errors

Exceptions in C++ can either be objects or basic types of member functions and variables. You can write a few lines of code by creating a hierarchy of certain exception objects. You can group these objects based on their classes and namespaces and then categorize them based on their type.

Exception Handling Examples

The following is an example of how you can use exception handling in C++. This program's output will explain how the try-catch blocks of code are used by the compiler.

```
#include <iostream>
using namespace std;
int main()
{
    int x = -1;
    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

The following is the output of the code:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

C++ also allows you to use a specific type of exception handler called the ‘catch-all’ block. You can use this function to catch any exceptions in the code. For instance, the following program throws an exception as an integer value. There is, however, no way to catch the error. In this case, the catch(…) block of code will be executed by the compiler.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
        throw 10;
     }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

The output of the following code is: Default Exception.

As mentioned in the first book, you cannot convert primitive data types in the code. This means you cannot use implicit type conversions in the code. Consider the following example where we are trying to convert a character into an integer.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

You receive the following output when you run the code: Default Exception.

If the compiler throws an exception that is not caught anywhere in the code, the program will end. For instance, the following program throws a char exception since there is no catch block or exception handling code to catch this error.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```

The output of the above code is:

```
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an

unusual way. Please contact the application's support team for

more information.
```

You can change the way the code behaves by writing blocks of code to indicate unexpected functions.

There are instances when you may receive an error in the derived class block of code. It is important to ensure the compiler can catch the error in this code before it looks at the base class exception. Therefore, you should first write exception handling code in the derived class. C++, like Java, is built with a library of exceptions that caters to all forms of base or standard exceptions. Any standard exception in your code can be caught by the compiler using this type.

It is unfortunate that the exceptions in C++ are left unchecked by the compiler. The compiler does not care if the exception in the code is identified or not. C++ does not require the compiler to specify the exceptions caught in the code or program. It is recommended that the exceptions are identified. Consider the following example. You receive an output with no exceptions, but the function fun() should have listed a set of unchecked exceptions.

```
#include <iostream>
using namespace std;
// This function signature is fine by the compiler, but not
recommended.
// Ideally, the function should specify all uncaught exceptions and
function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
The output: Caught exception from fun().
```

It is better to write the above code in the following manner:

```
#include <iostream>
using namespace std;
// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
```

```
            throw x;
        /* Some functionality */
    }
    int main()
    {
        try {
            fun(NULL, 0);
        }
        catch(...) {
            cout << "Caught exception from fun()";
        }
        return 0;
    }
```

The output of the above code is: Caught exception from fun().

You can also use try-catch blocks and nest them in the code if you want to re-throw exceptions in the output:

```
    #include <iostream>
    using namespace std;
    int main()
    {
        try {
            try {
                throw 20;
            }
            catch (int n) {
                cout << "Handle Partially ";
                throw; // Re-throwing an exception
            }
        }
        catch (int n) {
            cout << "Handle remaining ";
        }
        return 0;
    }
```

The output is: Handle Partially Handle remaining

You can also use the ‘throw’ keyword if you want to re-throw an exception. The function can then handle a part of the exception and let the caller handle the other part. When the code throws an exception, every object created in the class or try block will be removed from the code before the compiler moves back to the code's catch block.

```
#include <iostream>
using namespace std;
class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};
int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

The following is the output of the code:

```
Constructor of Test
Destructor of Test
Caught 10
```

Chapter 18

Stack Unwinding

You can remove functions in the code using an option known as stack unwinding. Using stack unwinding, you can remove function arguments, parameters, operators, and objects from the function's call stack. Stack handling is closely related to exception handling. When the compiler identifies an error in the code, it will look at the entire call stack to identify the error. The compiler looks for the exception handler in the code and the different entries associated with the handler. It will then remove the entire function with the exception handler code block from the function stack. This indicates that stack unwinding is a part of the exception handling process, especially since it relates to a function that is not handled in the same function.

Consider the following program:

```
#include <iostream>


using namespace std;


// a sample function f1() that throws an int exception
void f1() throw (int) {
  cout<<"\n f1() Start ";
  throw 100;
  cout<<"\n f1() End ";
}


// Another sample function f2() that calls f1()
```

```
void f2() throw (int) {
  cout<<"\n f2() Start ";
  f1();
  cout<<"\n f2() End ";
}


// Another sample function f3() that calls f2() and handles exception
thrown by f1()
void f3() {
  cout<<"\n f3() Start ";
  try {
    f2();
  }
  catch(int i) {
   cout<<"\n Caught Exception: "<<i;
  }
  cout<<"\n f3() End";
}


// a driver function to demonstrate Stack Unwinding  process
int main() {
  f3();


  getchar();
  return 0;
}
```

The following is the output of the code:

```
f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End
```

Let us look at the different functions in the code. When the first function [f1()] throws an exception in your output, the entry is removed from the call stack. The compiler does this since the function with the error does not contain an exception handler code. The second function [f2()] does not have an exception handler code in it. Therefore, the compiler will remove the code from the stack. The last function in the code is [f3()]. The code in this function contains an exception handler block of code. When the compiler identifies the error, it will execute the catch block in the code. The compiler will run the code after the exception handler code. It is important to note that the compiler does not run the lines of code found in the first two functions.

```
//inside f1()
  cout<<"\n f1() End ";
//inside f2()
  cout<<"\n f2() End ";
```

If you have some local objects created in the first two functions, the destructors used in the base functions will be called to remove any lines or blocks of code during the process of stack unwinding.

Chapter 19

Identifying Exceptions in Base and Derived Classes

If the compiler identifies exceptions in both the base and derived classes in the code, then it will not run the code or give you an output. When you write the code, you need to write an exception handler code in both the base and derived classes, and the exception handler block should be present before the block for the base class.

You may wonder why you need to add the exception handler block of code for the derived class before the base class. If you write the code for the base class before the derived class, the compiler will not reach the exception handler block of code in the derived class. For instance, the program below will give you the output: “Caught Base Exception.”

```
#include<iostream>
using namespace std;
  class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
```

```
    catch(Derived d) {  //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

If you change the order of the exception handler statements in the code above, the compiler can access both the statements easily. The following is a modified program of the above code, but it prints the following output: “Caught Derived Exception.”

```
#include<iostream>
using namespace std;
  class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
        // Some monitored code
        throw d;
    }
    catch(Derived d) {
        cout<<"Caught Derived Exception";
    }
    catch(Base b) {
        cout<<"Caught Base Exception";
    }
    getchar();
    return 0;
}
```

In C++, the compiler may throw an error if it catches or goes through the exception handler code in the base class before it looks at the derived class. It will, however, continue to compile the code. Java, another object-oriented programming language, does not throw any exceptions or errors if this happens. Consider the example below. This code returns the following output: “exception Derived has already been caught.”

```
//filename Main.java
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
        throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
}
```

Differentiating Between Block and Type Conversions

Consider the following code:

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


int main()
{
    try
    {
        throw 'x';
    }
    catch(int x)
    {
        cout << " Caught int " << x;
    }
    catch(...)
    {
        cout << "Default catch block";
    }
}
```

The output of this block of code is: “Default catch block.”

In the program above, an exception is thrown by the compiler in the form of a character. There is an exception handler block of code, but this is only to catch an int error in the code. You may think that the compiler will match the character's ASCII code and use the int exception handler to take care of the error, but this is not what happens in C++. Consider the following example where the exception handler code is not called for the object thrown as an error.

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


class MyExcept1 {};
class MyExcept2
{
public:
     // Conversion constructor
    MyExcept2 (const MyExcept1 &e )
    {
        cout << "Conversion constructor called";
    }
};
int main()
{
    try
    {
        MyExcept1 myexp1;
        throw myexp1;
    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(...)
    {
        cout << " Default catch block " << endl;
    }
    return 0;
}
```

Chapter 20

Object Destruction and Error Handling

Before you run the code in C++, try to predict the output of the following code:

```
#include <iostream>
using namespace std;
  class Test {
public:
  Test() { cout << "Constructing an object of Test " << endl; }
  ~Test() { cout << "Destructing an object of Test "  << endl; }
};
  int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

The output of this code is:

```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

When the compiler throws an exception, the code's destructor functions will be called to remove the objects whose scope ends with the entire block. This destructor is called before the compiler executes the exception handler code. For this reason, the code above gives you the output “Caught 10” after

“Destructing an object of Test.” How do you think the compiler will act when it identifies an exception in the constructor? Consider the following example:

```
#include <iostream>
using namespace std;
  class Test1 {
public:
  Test1() { cout << "Constructing an Object of Test1" << endl; }
  ~Test1() { cout << "Destructing an Object of Test1" << endl; }
};
  class Test2 {
public:
  // Following constructor throws an integer exception
  Test2() { cout << "Constructing an Object of Test2" << endl;
            throw 20; }
  ~Test2() { cout << "Destructing an Object of Test2" << endl; }
};
  int main() {
  try {
    Test1 t1;  // Constructed and destructed
    Test2 t2;  // Partially constructed
    Test1 t3;  // t3 is not constructed as this statement never gets
executed
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

The output of the above program is:

```
Constructing an Object of Test1
Constructing an Object of Test2
Destructing an Object of Test1
Caught 20
```

The compiler calls the destructor only when it uses completely constructed objects. If the constructor of the object leaves an exception, the compiler does not call for the destructor. Before you execute the following program, try to predict its outcome.

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


class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
        throw 4;
  }
  ~Test() { cout << "Destructing object number " << id << endl; }
};


int Test::count = 0;


int main() {
  try {
    Test array[5];
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
```

Chapter 21

Searching Algorithms

A searching algorithm is designed to look for an element or print the same element from the program's data structures or variables. There are numerous algorithms you can use to search for elements in the structures. The algorithms are classified into the following categories:

- *Interval search*: An interval search is one where the algorithm will look for the element in a sorted structure. These algorithms are better to use when compared to the next category since the structure is broken down and divided into parts before the element is identified in the structure—for example, binary search.
- *Sequential search*: In these types of algorithms, the compiler moves from one element to the next to look for the element in the data structure. An example of this algorithm is linear search.

Let us look at how these search algorithms work in C++.

Linear Search

Let us understand how the search algorithm works in C++ using an example. Consider a problem where you have an array ‘arr[]’ with n elements; how would you look for the value ‘x’ in the arr[]?

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                     110, 100, 130, 170}
          x = 110;
```

Output : 6

```
Element x is present at index 6
Input : arr[] = {10, 20, 80, 30, 60, 50,
                110, 100, 130, 170}
        x = 175;
```

Output : -1

Element x is not present in arr[].

The simplest way to perform a linear search is as follows:

1. Begin at the end of the array and compare the element you are looking for against each array element.
2. If the element matches one of the elements in your array, return the index
3. If the element does not match, move to the next element
4. If the element is not present in the array, return -1.

```
// C++ code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1


#include <iostream>
using namespace std;


int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}


// Driver code
int main(void)
```

```
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);


    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Binary Search

As mentioned earlier, a binary search is based on the interval search algorithm, where you look for an element in a sorted array. When compared to a linear search algorithm, a binary search algorithm has a higher time complexity.

A binary search uses the whole array as the interval when the search starts. It then breaks the interval into parts to look for the search element. It divides the array into half and looks for the element in the array's lower and upper sections. Depending on where the element lies, the algorithm will break the interval into a smaller section to look for it. It continues to do this until it finds the element.

A binary search aims to use the existing information in the array after it sorts the elements. This reduces the time complexity of the algorithm to O (log n). In a binary search, half the elements are not considered after making one comparison.

1. Sort the array.
2. Compare the search element x with the element in the middle of the array.
3. If x is less than the middle element, ignore the right section of the array since x can only lie in the left section of the array.

4. We then perform the same functions with the left section of the array.
5. If x is greater than the middle element in Step 3, we consider the array's right section.

We will now look at two ways to implement the binary search algorithm: recursive and iterative.

Before that, let us understand the time complexity of the binary search algorithm. You can calculate the time complexity of an algorithm using the following formula: T(n) = T(n/2) + c

You can remove the recurrence by using a master or recurrence tree method.

Recursive Implementation

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;


// a recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;


        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;


        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
```

```
                return binarySearch(arr, l, mid - 1, x);


            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, r, x);
        }


        // We reach here when element is not
        // present in array
        return -1;
    }


    int main(void)
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;
        int n = sizeof(arr) / sizeof(arr[0]);
        int result = binarySearch(arr, 0, n - 1, x);
        (result == -1) ? cout << "Element is not present in array"
                       : cout << "Element is present at index " << result;
        return 0;
    }
```

The output of the code is:

```
Element is present at index 3
```

Iterative Implementation

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;


// a iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
```

```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;


        // Check if x is present at mid
        if (arr[m] == x)
            return m;


        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;


        // If x is smaller, ignore right half
        else
            r = m - 1;
    }


    // if we reach here, then element was
    // not present
    return -1;
}


int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

The output of the code is: Element is present at index 3

Jump Search

The jump search algorithm is similar to the binary search algorithm in the sense that it looks for the search element in a sorted array. This algorithm's objective is to search for the search element from fewer elements in the array. The compiler can jump ahead by skipping a few elements or jumping ahead by a few steps.

Let us understand this better using an example. Suppose you have an array with n elements in it, and you need to jump between the elements by m fixed steps. If you want to look for the search element in the array, you begin to look at the following indices a[0], a[m], a[2m], ….. a[km]. When you find the interval where the element may be, the linear search algorithm kicks in.

Consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). The length of this array is 16. Let us now look for element 55 in the array. The block size is 4, which means the compiler will jump four elements each time.

Step 1: The compiler moves from index 0 to 3.
Step 2: The compiler moves from 4 to 8.
Step 3: The compiler jumps from 9 to 12.
Step 4: The element in position 12 is larger than 55, so we go back to the previous step.
Step 5: The linear search algorithm kicks in and looks for the index of the element.

Optimal Block Size

When you use the jump search algorithm, you need to choose the right block size, so there are no issues in the algorithm. In the worst-case scenario, you need to perform n/m jumps. If the element the compiler last checked was greater than the search element, you need to perform m-1 comparisons when the linear search algorithm kicks in. Therefore, in the worst-case scenario, the number of jumps should be ((n/m) + m-1). This function's value will be minimum if the value of the element ‘m’ is square root n. The step size, therefore, should be m = √n.

```
// C++ program to implement Jump Search

#include <bits/stdc++.h>
using namespace std;

int jumpSearch(int arr[], int x, int n)
{
    // Finding block size to be jumped
    int step = sqrt(n);

    // Finding the block where element is
    // present (if it is present)
    int prev = 0;
    while (arr[min(step, n)-1] < x)
    {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }

    // Doing a linear search for x in block
    // beginning with prev.
    while (arr[prev] < x)
    {
        prev++;

        // If we reached next block or end of
        // array, element is not present.
        if (prev == min(step, n))
            return -1;
    }
    // If element is found
    if (arr[prev] == x)
        return prev;


    return -1;
}


// Driver program to test function
int main()
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
                34, 55, 89, 144, 233, 377, 610 };
    int x = 55;
    int n = sizeof(arr) / sizeof(arr[0]);


    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x, n);


    // Print the index where 'x' is located
    cout << "\nNumber " << x << " is at index " << index;
    return 0;
}
```

The output of this code: Number 55 is at index 10

Some important points to note about this algorithm are:

- This algorithm only works when an array is sorted
- Since the optimal length the compiler should jump is √ n, the time complexity of this algorithm is O (√ n). This means the time complexity of this algorithm is between the binary search and linear search algorithms

The jump search algorithm is not as good as the binary search algorithm, but it is better than the binary search algorithm since the compiler only needs to move once back through the array. If the binary search algorithm is too expensive, you should use the jump search algorithm.

Chapter 22

Sorting Algorithms

You use sorting algorithms when you want to arrange a list or array of elements based on the comparison operator you want to include. This comparison operator will be used to decide how the elements will be sorted in the data structure where you want to arrange the elements. Let us look at some common sorting algorithms.

Bubble Sort

The bubble sort algorithm is one of the simplest algorithms used in C++, and it works by swapping the elements adjacent to each other in the array in case they are not in the right order.

Consider the following example:

First Pass

(5 1 4 2 8) –> (1 5 4 2 8): In this step, the algorithm will compare the elements in the array and swap the numbers 1 and 5.

(1 5 4 2 8) –> (1 4 5 2 8): In this step, the numbers 4 and 5 are swapped since the number 5 is greater than 4.

(1 4 5 2 8) –> (1 4 2 5 8): In this step, the numbers 5 and 2 are swapped.

(1 4 2 5 8) –> (1 4 2 5 8): In the last step, the elements are ordered, so there is no more swapping necessary.

Second Pass

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8): In this step, the numbers 4 and 2 are swapped since the number 4 is greater than 2.

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

The array is already sorted, but since the compiler is not aware that the algorithm is complete, it will complete another round on the array and check if any elements need to be swapped.

Third Pass

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Consider the following implementations of the bubble sort algorithm:

```
// C++ program for implementation of Bubble sort
#include <bits/stdc++.h>
using namespace std;


void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}


// a function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}


/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}


// Driver code
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

The output of this code is:

```
Sorted array:

        11 12 22 25 34 64 90
```

We can optimize the implementation of this sorting algorithm. The above code runs more number of times than necessary, although the array is sorted. You cannot stop the optimization since the inner loop does not perform any swaps.

```
// Optimized implementation of Bubble sort
#include <stdio.h>


void swap(int *xp, int *yp)
{
     int temp = *xp;
     *xp = *yp;
     *yp = temp;
}


// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(&arr[j], &arr[j+1]);
           swapped = true;
        }
     }


     // IF no two elements were swapped by inner loop, then break
     if (swapped == false)
        break;
   }
}


/* Function to print an array */
```

```
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}


// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

The output of the above code is:

```
Sorted array:
        11 12 22 25 34 64 90
```

Selection Sort

Using the selection sort algorithm, you can sort the elements in the array by looking at the smallest element in the array in ascending order. The compiler will only perform this sorting algorithm in the part of the array with unsorted elements. The algorithm divides the array into two arrays:

- The section of the array with sorted elements
- The section of the array which does not have any sorted elements

When the selection sort algorithm iterates, the element which is the smallest in the array from the unsorted section of the array. This element is then moved to the sorted section of the array. Consider the following section:

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at the beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at the beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at the beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Let us write down the above example in the form of a program:

```
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;


void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}


void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;


        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}


/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}


// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

The output of the program is:

```
Sorted array:
```

```
11 12 22 25 64
```

Insertion Sort

The insertion sort algorithm is a simple and straightforward algorithm that works in the same way you would sort or arrange playing cards. In this algorithm, the array is split into two parts – sorted and unsorted arrays. The values in the unsorted array will then be sorted and moved into the right positions.

Let us understand this algorithm better by considering how it works when the compiler tries to sort the elements in the ascending order:

1. Create an array with n elements.
2. Iterate from the first element in the array to the nth element in the array
3. Compare every element in the array with its predecessor
4. If the element is smaller than the predecessor, you should compare it to the other elements before the predecessor. Move the elements around to ensure there is enough space for the element which is swapped

Consider the following example: The array is 12, 11, 13, 5, 6

12, 11, 13, 5, 6

We will add a loop where the iterative element ‘i’ is assigned the value 1. Since there are only 5 elements in the array, the value of ‘i' can be incremented until 4.

i = 1. Since element 11 is smaller than 12, the number 12 is moved after element 11.

11, 12, 13, 5, 6

i = 2. The number 13 will stay in its position since the first three elements are sorted.

11, 12, 13, 5, 6

i = 3. The number 5 will move to the start of the array since it is the smallest number when compared to the other elements in the array. The other elements will be moved ahead.

5, 11, 12, 13, 6

i = 4. The number 6 will move next to 5 since it is smaller than all the other elements in the array.

5, 6, 11, 12, 13

Let us write the above example in C++.

```
// C++ program for insertion sort
#include <bits/stdc++.h>
using namespace std;


/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;


        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
// a utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}


/* Driver code */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);


    insertionSort(arr, n);
    printArray(arr, n);


    return 0;
}
```

Quicksort

The quicksort algorithm is a divide and conquer algorithm. In this algorithm, the compiler picks an element in the array as the pivot and divides the elements in the array based on that pivot. There are different ways to implement the quick sort algorithm in C++.

1. The algorithm can choose the first element in the array as the pivot
2. The algorithm can choose the last element as the pivot (we will look at this in detail in the section below)
3. Choose any element in the array as the pivot
4. Choose the median of the elements in the array and pick that as

the pivot

One of the most important processes in a quick sort algorithm is the partition() function. This function's target is to take an element from the array as the pivot and put it in the right position. The elements in the array which are smaller than the pivot will be moved to one side of the array while the others will move to the other section of the pivot. Consider the following pseudo-code for this algorithm:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Understanding the Partition Algorithm

```
/* low  --> Starting index,  high  --> Ending index */

quickSort(arr[], low, high)

{

    if (low < high)

    {

        /* pi is partitioning index, arr[pi] is now

           at right place */

        pi = partition(arr, low, high);



        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi

    }

}
```

The pseudo code for the partition algorithm is:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];


    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Let us look at the illustration of this function:

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped
```

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Let us look at how to implement this algorithm in C++:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;


// a utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}


/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element


    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}


/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);


        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}


// Driver Code
int main()
```

```
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Chapter 23

Tips to Optimize Code in C++

When you write code in C++ or any other programming language, your main objective should be to write code that works correctly. Once you accomplish this, you need to change the code to improve the following:

1. The security of the code
2. The quantity of memory used while running the code
3. Performance of the code

This chapter gives you a brief idea of the areas to consider if you want to improve the performance of your code. Some points to keep in mind are:

- You can use numerous techniques to improve the performance of your code. This method, however, can lead to the creation of a larger file.
- If you choose to optimize multiple areas in your code at the same time, it may lead to some conflict between the areas of your code. For instance, you may not be able to optimize both the performance of the code and memory use. You need to strike a balance between the two.
- You may always need to optimize your code, and this process is never-ending. The code you write is never fully optimized. There is always room to improve some parts of your code if you want the code to run better.
- You can use different tricks to improve the performance of the code. While you do this, you should ensure that you do not forget about

some coding standards. Therefore, do not use cheap tricks to make the code work better.

Using the Appropriate Algorithm to Optimize Code

Before you write any code, you need to sit down and understand the task. You then need to develop the right algorithm to use to optimize the code. We are going to understand how the algorithm affects your code using a simple example. In the program, we are going to use a two-dimensional segment to identify the maximum value and, for this, we will take two whole numbers. In the first code, we will not look at the program’s performance. We will then look at a few methods to use to improve the performance of the code.

Consider the following parameters used in the code: both numbers should lie between the interval [-100, 100]. The maximum value is calculated using the function: (x * x + y * y) / (y * y + b).

There are two variables used in this function – x and y. We are also using a constant ‘b’ which is a user-defined value. The value of this constant should always be greater than zero but less than 1000. In the example below, we do not use the pow() function from the math.h library.

```
#include <iostream>
#define LEFT_MARGINE_FOR_X -100.0
#define RIGHT_MARGINE_FOR_X 100.0
#define LEFT_MARGINE_FOR_Y -100.0
#define RIGHT_MARGINE_FOR_Y 100.0
using namespace std;
int
main(void)
{
//Get the constant value
cout<<"Enter the constant value b>0"<<endl;
cout<<"b->"; double dB; cin>>dB;
if(dB<=0)   return EXIT_FAILURE;
if(dB>1000) return EXIT_FAILURE;
//This is the potential maximum value of the function
//and all other values could be bigger or smaller
double dMaximumValue =
(LEFT_MARGINE_FOR_X*LEFT_MARGINE_FOR_X+LEFT_MAR
(LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y+dB);
double dMaximumX = LEFT_MARGINE_FOR_X;
double dMaximumY = LEFT_MARGINE_FOR_Y;
for(double dX=LEFT_MARGINE_FOR_X;
dX<=RIGHT_MARGINE_FOR_X; dX+=1.0)
  for(double dY=LEFT_MARGINE_FOR_Y;
dY<=RIGHT_MARGINE_FOR_Y; dY+=1.0)
    if( dMaximumValue<((dX*dX+dY*dY)/(dY*dY+dB)))
    {
      dMaximumValue=((dX*dX+dY*dY)/(dY*dY+dB));
      dMaximumX=dX;
      dMaximumY=dY;
    }
cout<<"Maximum value of the function is="<<
dMaximumValue<<endl;
cout<<endl<<endl;
cout<<"Value for x="<<dMaximumX<<endl
    <<"Value for y="<<dMaximumY<<endl;
  return EXIT_SUCCESS;
}
```

Look at the code carefully. You notice that the function and value dX * dX is run by the process too many times, and the value is stored multiple times in the memory. This is a waste of CPU time and memory. What do you think we could do to improve the speed of the code? An alternative to writing the operation multiple times in the code is to declare a variable and assign this function to it. Let us define a variable d, which stores the value of the function dX * dX. You can use the variable ‘d’ everywhere in the code where you need to use the calculation. You can optimize other sections of the above code as well. Try to spot those areas.

The next area we need to look at is how general the lines of code are. You need to see whether the program runs as fast as you want it to. If you want to increase the speed of the algorithm, you need to tweak some functions based on the size of your input. What does this mean?

You can improve the speed of the code you have written using multiple algorithms instead of only one algorithm. When you use two algorithms, you can instruct the compiler to switch between the algorithms based on a condition.

Optimizing Code

When you write code, every element in your code uses some space in the memory. It is important to understand how each word in your code uses memory to reduce consumption or usage. Let us consider a simple example where we try to swap the values in two variables in the memory. You can do this using numerous sorting algorithms. To understand this better, let us take a real-world example – you have two people sitting in two different chairs. You introduce a third or temporary chair to hold one of the individuals when they want to swap chairs.

Consider the following code:

```
int nFirstOne =1, nSecondOne=2;
int nTemp = nFirstOne;
nFirstOne = nSecondOne;
nSecondOne = nTemp;
```

This code is easy to use, but when you create a temporary variable in your code, the compiler will assign some space in the memory for this object. You can avoid wasting memory space by avoiding the usage of a temporary variable in the code.

```
int nFirstOne = 3, nSecondOne = 7;
nFirstOne += nSecondOne;
nSecondOne = nFirstOne ? nSecondOne;
nFirstOne -= nSecondOne;
```

You may need to swap large values in the memory to a different section. How would you do this? The easiest way to do this is to use pointers. Instead of copying the same value across the memory, use a pointer to obtain the address of the value in the memory. You can then change their address instead of moving the value from one location to the next in the memory.

You may wonder how you can determine if your code is faster or how you can calculate this. When you finish writing your code, the system will translate it into a language it understands using the assembler. It will then translate this into machine code, which it quickly interprets. Every operation you write in the code takes place in the processor. It may also take place in the graphic card or mathematical coprocessor.

One operation can take place in one clock cycle, or it may take a few. For this reason, it is easier for the computer to multiply numbers compared to division. This could be the case because of the optimization the computer performs. You can also leave the task of optimization to the compiler in some cases.

If you want to learn more about how fast your code is, you should know the architecture of the computer you are using. The code can be faster because of one of the following reasons:

1. The program runs in the cache memory
2. The mathematical coprocessor processes sections of the code
3. A branch predictor was used correctly by the compiler

Let us now consider the following numbers: O(n), O(log(n) *n), n*n, n!. When you use this type of code, the program's speed depends on the number you key into the system. Let us assume you enter n = 10. The program may take ‘t’ amount of time to run and compile. What do you think will happen when you enter n = 100? The program may take 10 times longer to run. It is important to understand the limits a small number can have on your algorithm.

Some people also take time to see how fast the code runs. This is not the right thing to do since not every program or algorithm you key in is completed first by the processor. Since an algorithm does not run in the computer’s kernel mode, the processor can get another task to perform. This means the algorithm is put on hold. Therefore, the time you write down is not an accurate representation of how fast the code can run. If you have more than one processor in the system, it is harder to identify which processor is running the algorithm. It is tricky to calculate the speed at which the processor completes running your code.

If you want to optimize or improve the speed at which the program runs, you need to prevent the processor from shifting the code to a different core during the run. You also need to find a way to prevent the counter from switching between tasks since that only increases the time the processor takes to run the code. You may also notice some differences in your code since the computer does not transfer all optimizations into machine code.

Using Input and Output Operators

When you write code, it is best to identify the functions you can use which do not occupy too much space in the memory. Most times, you can improve the speed of the program by using a different function to perform the same task. Printf and scanf are two functions used often in C programming, but you can use the same keywords in C++ if you can manipulate some files. This increases the speed of the program and can save you a lot of time and memory.

Let us understand this better through an example. You have two numbers in a file and need to read those numbers. It is best to use the keywords cin and cout on files in terms of security since you have instructions passed to the compiler from the header library in C++. If you use printf or scanf, you may need to use other functions of keywords to increase the speed of the program. If you want to print strings, you can use the keyword put or use an equivalent from file operations.

Optimizing the Use of Operators

You need to use operators to perform certain functions in C++. Basic operators, such as +=, *= or -= use a lot of space in the memory. This is especially true when it comes to basic data types. Experts recommend you use a postfix decrement or increment along with the prefix operator if you want to improve the functioning of the code. You may also need to use the << or >> operators instead of division and multiplication, but you need to be careful when you use those operators. This may lead to a huge mistake in the code. It takes some time to identify these mistakes and, to overcome the mistake, you need to add more lines of code. This is only going to reduce the speed and performance of the program.

It is best to use bit operators in your code since these increase the speed of the program. If you are not careful about how you use these operators, you may end up with machine-dependent code, and this is something you need to avoid.

C++ is a hybrid language and allows you to use an assembler's support to improve the functioning of your program. It also allows you to develop solutions to problems using object-oriented programming. If you are adept at coding, you can develop libraries to improve your code's functioning.

Optimization of Conditional Statements

You may need to include numerous conditional statements in your code, depending on the type of code you are writing. Most people choose to use the 'if' conditional statement, but it is advised that you do not do this. It is best to use the switch statement. When you use the former conditional statement, the compiler needs to test every element in the code, and this creates numerous temporary variables to store the code. This reduces the performance of the code.

It is important to note that the 'if' conditional statement has many optimizations built into the statement itself. If you only have a few conditions to test, and if these are connected to the or operator, you can use the 'if' conditional statement to calculate the value. Let us look at this using an example. We have two conditions, and each of these uses the and operator. If you have two variables and want to test if both values are equal to a certain number, you use the and operator. If the compiler notes that one value does not meet the condition, it returns false and does not look at the second value.

When you use conditional statements in your code, it is best to identify the statements which often occur before the other conditional statements. This is the best way to determine if an expression is true or false. If you have too many conditions, you need to sort them and split them into a nested conditional statement. There may be a possibility that the compiler does not look at every branch in the nest you have created. Some lines of code may be useless to the compiler, but they simply occupy memory.

You may also come across instances where you have long expressions with numerous conditions. Most programmers choose to use functions in this instance, but what they forget is that functions take up a lot of memory. They create calls and stacks in memory. It is best to use a macro to prevent the usage of memory. This increases the speed of the program. It is important to remember that negation is also an operation you can use in your code.

Dealing with Functions

If you are not careful when you use functions, you may end up with bad code. Consider the following example. If you have code written in the same format as the statements below, it will lead to a bad code:

```
for(int i=1; i<=10; ++i)
      DoSomething(i);
```

Why do you think this is the case? When you write some code similar to the above, you need to call the function a few times. It is important to remember that the calls the compiler makes to the functions in the code use a lot of memory. If you want to improve the performance of the code, you can write the statement in the following format:

```
DoSomething(n);
```

The next thing you need to learn more about is inline functions. The compiler will use an inline function similar to a macro if it is small. This is one way to improve the performance of your code. You can also increase the reusability of the code in this manner. When you pass large objects from one function to another, it is best to use references or pointers. It is better to use a reference since this allows you to write code that is easy to read. Having said that, if you are worried about changing the value of the actual variable being passed to the function, you should avoid using references. If you use a constant object, you should use the keyword const since it will save you some time.

It is important to note that the arguments and parameters passed in the function will change depending on the situation. When you create a temporary object for a function, it will only reduce the speed of the program. We have looked at how you can avoid using or creating temporary variables in the code.

Some programmers use recursive functions depending on the situation. Recursive functions can slow the code down. So you should avoid the use of recursive functions if you can since these reduce the performance of your code.

Optimizing Loops

Let us assume you have a set of numbers, and you are to check if the value is greater than 5 or less than 0. When you write the code, you need to choose the second option. It is easier for the compiler to check if a value is greater than zero than to check which number is greater than 10. In simple words, the statement written below makes the program slower when compared to the second statement in this section.

```
for( i =0; i<10; i++)
```

As mentioned, it is best to use this loop instead of the above. If you are not well-versed with C++ programming, this line of code may be difficult for you to read.

```
for(i=10; i--; )
```

Similarly, if you find yourself in a situation where you need to pick from <=n or !=0, you should choose the second option since that is faster. For instance, if you want to calculate a factorial, do not try to use a loop since you can use a linear function. If you ever find yourself in a situation where you need to choose between a few loops or one loop with different tasks, you should choose the second option. This method may help you develop a better performing program.

Optimizing Data Structures

Do you think a data structure affects the performance of your code? It is not easy to answer this question. Since data structures are used everywhere in your code, the answer is difficult to formulate and vague. Let us look at the following example to understand this better. If you are tasked with creating permutations (using the pattern below), you may choose to use a linked list or array.

```
1, 2, 3, 4,
2, 3, 4, 1,
3, 4, 1, 2,
4, 1, 2, 3,
```

If you use an array, you can copy the first element in the array and move every other element in the array towards that element. You then need to move the first element in the array to the end of the list. To do this, you need to use multiple operations, and your program will be very slow. If you leave the data in a list, you can develop a program, which will improve the performance of the code. You can also store the data in the form of a tree. This data structure allows you to develop a faster program.

Bear in mind that the type of data structure you use affects the performance of your program. You can solve any problem you have in the code without using arrays or any other data structure.

Sequential or Binary Search?

When you look for a specific object or variable in the code, which method should you opt for – binary or sequential search?

No matter what you do in your code, you always look for some value in a data structure. You may need to look for data in tables, lists, etc. There are two ways to do this:

1. The first method is simple. You create an array and assign some values to the array. If you want to look for a specific value in the array, you need to start looking at the start of the array until you find the value in the array. If you do not find the value at the start of the array, the compiler moves to the end of the array. This reduces the speed at which the program is compiled.
2. In the second strategy, you need to sort the array before you search for an element in the array. If you do not sort the array before you look for the element, you cannot obtain the results on time. When the array is sorted, the compiler will break it into two parts from the middle. It will then look for the value in either part of the array depending on the values in the sections. When you identify the part where the element may be, you need to divide it through the middle again. You continue to do this until you find the value you are looking for. If you do not, then you know the array does not have the value.

What is the difference between these strategies? When you sort the elements in the array, you may lose some time. Having said that, if you give the compiler time to do this, you will benefit faster from the search. When it comes to choosing between a sequential and binary search, you need to understand the problem before you implement the method you want to use.

Optimizing the Use of Arrays

We looked at arrays in the previous book, and this is one of the basic data structures used in C++. An array contains a list of objects of a similar data type. Every object in an array holds a separate location in the memory.

If you want to learn more about optimizing the work or use of an array, you need to understand the structure of this structure. What does this mean? An array is similar to a pointer, and it points to the elements in the array. You can access array methods using arithmetic pointers or any other type of pointer if needed. Consider the following example:

```
for(int i=0; i<n; i++) nArray[i]=nSomeValue;
```

The code below is better than the statement above. Why do you think this is the case?

```
for(int* ptrInt = nArray; ptrInt< nArray+n; ptrInt++)
*ptrInt=nSomeValue;
```

The second line of code is better than the first line of code since the operations rely only on pointers. In the example above, we are using pointers to access the values stored in the integer data type. The pointer takes the address of the variable in the memory. In the case of the example, the pointer points to the variable nArray. When we add the increment operator to the variable, the pointer will move from the first element in the array to the next until it reaches the end of the array. If you use the double data type, the compiler will know how far it should move the address.

It is difficult to interpret and read the code using this method, but this is the only way to increase the speed of the program. In simple words, if you do not use a good algorithm, you can increase compiling speed by writing code using the right syntax.

Consider the following example: You have a matrix with the required elements. A matrix is a type of array, and it will be stored in your memory based on the rows. So, how do you think you should access the elements in the array? You should access every element in the matrix row by row. It does not make sense for you to use any other method because you reduce the speed of the program.

It is best to avoid initializing large sections of the memory for only one element. If you know the size of the element, make sure to stick to that size. Do not allot more memory space. You can use the function memset or other commands to allot some space in the memory to the variables used in the code.

Let us assume you want to create an array of characters or strings. Instead of defining the variables or assigning the array to specific variables, it is best to use pointers. You can assign each element in the array to a string, but this would only reduce the speed of the program. The compiler will run the code faster, even if the file is big. If you use the new keyword to create or declare an array in the code, your program will not do well since it will use a lot of memory the minute you try to run the code. It is for this reason you should use vectors. These objects add some space to your memory, allowing the program to do well.

If you want to move large volumes of data from one section in the memory to another, it is best to use an array of pointers. When you do this, you do not change the original values of the data but only replace the addresses of the objects stored in the memory.

Chapter 24

Debugging and Testing

Before we look at the different aspects of debugging and testing, let us understand what these terms mean.

Definition

Testing

Testing is the process of identifying the behavior of the code and identifying the correct behavior. You can test the code at different stages of developing the code, including:

1. Module development
2. Requirements analysis
3. Interface design
4. Algorithm development
5. Implementation
6. Integration

In this chapter, we will look at what implementation testing is and how it helps to improve the functioning of the code. It is important to note that implementation testing does not mean you only test the code when you execute it. You can also perform this testing to check the correctness of the code used. Some programmers also use peer review to help them improve their code.

Debugging

It is important to note that debugging is an activity every coder must perform to ensure the code runs correctly. During the debugging process, you can correct any lines in the code which do not run correctly. Implementation testing is very different from debugging since the latter is used to locate any errors in the code, while the former is to test whether the code gives you the correct output. The testing strategies you implement during debugging and testing are based on this difference.

Conditions for Debugging

It is important to avoid spending too much time when you debug code. You, as a programmer, should be prepared. You need to put in a lot of effort to debug the code. Follow the steps given below to prepare yourself for the arduous task:

Understand the Algorithm and Design

It becomes very difficult to debug the code written if you do not understand the algorithm and design. You cannot test the module if you do not understand the design since you have no idea what the objective of the module is. If you do not understand the algorithm, you cannot locate any error in the code when you test it. Another reason why it is important to understand the algorithm is the test cases. If you do not know how the algorithm functions, you cannot develop effective test cases, and this is true when you use data structures in your code.

Check the Correctness of the Code

There are numerous methods used to check if the code is correct and runs smoothly.

Proof of Correctness

One of the easiest ways to check for any errors in the code is to examine every algorithm used in the code using some methods. For instance, if you know the invariants, preconditions, postconditions, and terminating conditions in a loop statement, then you can perform some easy checks in the code. Here are some questions you can ask to determine the correctness of the code:

1. If the compiler enters the loop, does this mean the invariant used

is true based on the precondition?

2. If the compiler moves through the loop, does it indicate that the loop is closer to termination?
3. If the loop is nearing the end, does it mean the compiler will move towards the postcondition?

Some of these checks may not indicate the errors in the code, but you will understand the algorithm better.

Code Tracing

It can be easy to detect some errors in the code by tracing how the modules or functions execute, especially when calls are made to the function or module in different parts of the program. Experts suggest that you, as the code writer, should trace the working of the modules and functions along with someone else. If you want this process to be effective, you should trace the modules and functions by assuming that other code work functions and procedures work accurately. You may need to deal with levels or layers of abstraction in the procedure and function. Tracing does not catch all errors, but it will improve your understanding of the algorithm used.

Peer Reviews

As the name suggests, peer review is asking a peer to examine and check your code for any errors. If you want the review to be effective, you must ensure the peer has the required information and knowledge to check the code. You may also need to give the peer the code in advance so they know what to read or expect.

As the code writer, you should meet with the reviewer and explain how the algorithms in the code work. If the reviewer disagrees or does not understand some parts of the implementation, you need to discuss it with him until you both reach an agreement. The reviewer’s objective should be to detect the errors in the code. You can then correct the errors identified.

You can identify or discover errors in your code when you review it. Having said that, it is useful if you have someone from the outside looking at the code and identifying some blind spots in the code. Peer reviews, like code tracing, may take some time. Make sure to restrict the reviews only to those sections in the code where you know there can be some problems.

Anticipate Errors

It is unfortunate that people make errors when they write code, and some arguments may not be called accurately by the functions and modules. We also make mistakes when it comes to tracing the code, and peer reviews may not catch all the errors in the code. Therefore, you need to be prepared for the errors your code may run into using the requirements mentioned below.

Debugging Requirements

You need to have two capabilities in your code when you try to debug your program. The first thing to do is call the functions and services used in the module, while the second thing to do is obtain the information about the results of those calls. You also need to learn more about the internal state and data available in the function or module.

How to Drive the Module

When you try to debug any module in your code, you should ensure there is a method available to call the services or functions used in the module. This can be done using the following methods:

Hardwired Drivers

Hardware drivers are sections in the main program which contain a sequence of calls to various functions in the program. You can modify the sequence of the calls by rewriting the code in the driver. If you want to test the modules which vary because of different variables, it is best to use a hardwired driver. These drivers cannot work with numerous cases, which is a shortcoming.

Command Onterpreters

When the compiler runs the code in the program, it uses a command interpreter to test the code by running the input statements and interpret the commands which execute the calls to functions and modules. You can design these interpreters so you can enter the command either from a file or interactively. It is best to use an interactive command interpretation, especially in the first stages of debugging, but it is best to use a batch mode in later stages. A disadvantage of using a command interpreter is that it is slightly complicated to write. You may also spend too much time debugging the code you have written for the interpreter. This disadvantage is overcome since most of the code can be used to test other sections of the module or function.

Learn More About the Module

When it comes to debugging, you need to learn more about the program's various modules and functions. It makes no sense to control the functions and their sequence if you do not have the information about the effect of those functions on your code's variables and data structures. If the statements provide an output, then you have enough information available. Having said that, most functions and statements in the code will change some aspects of the module. This would mean you need to learn more about the modules for debugging.

Module State

Most data structures and modules in C++ allow you to insert and delete data used in the module. These statements do not generate the necessary output because these statements do not return the information from the arguments and parameters entered in the functions. Therefore, if you want to debug or test any code in the module, include the right statements in the code to display the workings or changes made in the module. Most programmers add some procedures to display the various contents in the modules. The compiler uses the procedures while testing the code, but it removes them when the testing is complete. It is important to have the compiler show the internal structure of the modules during debugging.

Module Errors

If you develop modules with complex statements, it is hard to determine where the error has occurred. It is possible the compiler may call upon the wrong private subroutine. To avoid such errors, you need to write practical code.

Execution

To locate where the errors in the module are, it is important to know which subroutine or program the compiler used when the error occurred. If you want to know when the compiler is in the execution state, add print statements to the code to indicate when the compiler enters and exits some sections of the code.

Debugging Principles

Report Conditions

Most programmers spend a lot of time identifying the errors in the code. It is important to detect these errors early so that you identify the cause easily. If the compiler detects the error early on, it is important to find the cause. If the compiler detects the errors in the module early, you can identify the cause easily. If you detect the errors in the code only when the symptoms of the errors appear, it may be hard to identify the code.

Improve the Ease of Interpretation and Useful Information

It is important to maximize the information you obtain on executing debugging code. It is also important to learn to interpret the meaning of the information. As a programmer, you need to interpret the data and detect the error location in the code. You cannot rely on the error or debugging codes you write since each module relies on the entire structure. Therefore, it is important to display the entire structure in an easily understandable form.

Avoid Distracting and Meaningless Information

If you have too much information in the code, it can be a little overwhelming. If you have very little information, the error testing process is rendered useless. Let us assume you have a printout of all the times the compiler has entered and exited the code. When you look at this sheet, you cannot identify where the first error was identified. You should only use module execution reports only if you know where the error has occurred. As a rule, you should prefer code that tells you where the problem is and not that it is not in the code you are looking at.

Do Not Use Single-Use Testing Code

Most programmers make the mistake of using one single test code to check the error in the entire code. This code is extremely complex to write and understand. How would you feel if you were testing the debugging code, but there is an error in the debugging code itself? This is a waste of time. It is only practical to write complex test scripts if you know you can use different parts of the code in other debugging methods.

Functionalities to Use

Every programming language has some built-in functionalities you can use to debug the code.

Assertion Statements

Some C++ procedures have assertion procedures in the code that only take one parameter, often a Boolean statement. When you add a call to an assertion procedure in your code, the compiler executes the Boolean expression first. If the compiler executes this statement and the result is true, nothing additional is done but, when you receive a false, the compiler will end the execution and throw an error message. You can use this procedure to detect and report the error condition.

Tracebacks

Most compilers come with generic debugger codes that allow you to perform traceback operations. The compiler uses these, especially when there are runtime errors in your code. a traceback method of debugging gives you the list of active subroutines in the code. Tracebacks also give you the line numbers where you have active subroutines. If the error code identifies the sections where you have runtime errors, the traceback gives you the line numbers where the error has occurred. It is, however, up to you to identify which line in the subprogram caused the error.

Debugger Keywords

Most compilers and computers have different debugging programs or sections in the code. For instance, you can use the debugging keywords dbx and sdb in a Unix operating system. a debugging program gives you a way out – there is no need to develop a block of code for the sole purpose of debugging.

A debugging program runs through every line of code and identifies the errors in each line of the code. When the compiler executes a line with an error or break within it, the debugging code will create a break in the process, which allows the user to examine or modify the program data.

You can also use a debugging program or keyword to perform traceback operations in case you encounter any run-time errors. It is difficult to learn how a debugging code or keyword works. If this is the only tool you are using for debugging, then you may not save a lot of time. For instance, if you have a good debugger but a terrible test driver, the results of your debugging will not lead to the right results.

Techniques for Debugging

Incremental Testing

It is important to break the code into multiple subroutines if you are designing a complex module or function. It is important to note that the subroutines in the code should not be longer than 10 statements. If a module is designed this way, it is best to use incremental testing to work on your code.

When you perform incremental testing, you can classify subroutines as different levels. For instance, if you have a subroutine at a lower level, it means lower level subroutines do not call higher subroutines. Let us assume you have two subroutines – a and B. If a calls B, then B is a lower subroutine.

This form of testing aims to look at each subroutine as an individual block of code and test it. Begin with the lower subroutines. To do this, develop a test script that calls a lower subroutine directly. Otherwise, you need to include numerous test cases in your code, and each of these sections should include lower level subroutines.

To develop these test cases, you need to have a good understanding of the algorithms used in the module. The objective of this form of testing is to identify the errors in different sections of the code. This makes the process effective to debug the code and identify different sections in the code with errors.

Through incremental testing, you learn to work only with one error in the code. Most debugging and testing techniques look at multiple errors at once.

Sanity Checks

You can write a low-level code within a data structure under the assumption that the compiler can implement the code with the help of the high-level code. Most programmers write low-level code under the assumption that certain parameters or variables should not be null. The condition or assumption may be justified by the statements written in the method or program, but it is best to include a test script or block to see if the statements in the algorithm are implemented correctly. This is a sanity check, and the advantage of including these blocks of code is that the compiler can easily detect errors in the code.

Using Boolean Constants to Turn Off Debugging

If you want to include debugging code in your functions and methods, it is best to enclose those statements within a conditional statement. You can control the functioning of the conditional statement using a Boolean constant. This process allows you to turn the debugging code off easily, but the compiler can access it later if it needs to. It is best to use a different constant at every stage of the testing so that the compiler does not consider useless information.

Error Variables to Control the Program's Behavior

When you add debugging a print statement to your code, there is a possibility that the program may contain too much information which the compiler cannot use. The trouble with including such print statements is that the compiler will execute the statement regardless of whether there is an error or not. If the print statement is executed multiple times by the compiler, it indicates that it does not identify the error.

If the print statements display too much information of an existing data structure in the code, it indicates that the issue is magnified. If you do include a sanity check in your code to detect any errors, you should include a Boolean variable in the same module. Initialize the Boolean variable to false, which indicates there is no error in the code. You can create most data structures during the data initialization period, and you can initialize the error variable at the same time. Doing this will ensure that the compiler sets the error to true but does not exit the sanity check block of code. You can then enclose the debug code in a conditional statement, so the information in the block of code is printed only when the compiler identifies an error. You can do this using the next method.

Traceback Methods

If you want to use traceback to identify the error in your code, it is best to use Boolean sets to identify the error. It is easier to do this by performing a sanity check. Experts recommend that you add some error code in the functions or methods. This error code needs to be controlled by the variable causing the error. It is best to run the code and perform a sanity check before you add any error code to the program. You should add the debug code to control the errors in the program when the method or function calls on the sanity check.

How to Correct the Errors In Your Code

You need to keep the following objective in mind when you test code and identify any errors in your code – identify the cause and fix it. Do not worry about the symptoms.

Let us assume you have run the code, and you identify an issue with the segment. When you check the code carefully, you notice that you were passing a null pointer into a function or module. You, unfortunately, have not entered a statement to check if the pointer carries a null value. Does this mean you should check every method or function in the code for a null pointer? To do this, you may need to use an if statement and enclose the entire method or function in that if statement. You cannot determine this without understanding the design of the program and the algorithm you have used. The algorithm may have been implemented correctly in your code, and you can determine if the pointer is null using the algorithm. If this happens in your code, it means the if statement does not identify or solve the issue's cause.

When you add the if statement to the code, you only cover the indicators; you may hide the issue in one section of the code, but the issue will be elsewhere. It is important to note that you should include new code to your algorithm only if you are certain the statements should be added to the algorithm. Although the algorithm does not require it, if you still want to add a check, ensure it returns error codes and conditions. In simple words, you need to perform a sanity check.

Conclusion

If you have the basics of C++ down pat and want to learn more about programming in C++, you have come to the right place. This book has all the information you need about the language and how you can use object-oriented programming in C++. You will gather information about different concepts in object-oriented programming, such as abstraction, encapsulation, etc.

The book also introduces the concepts of searching and sorting algorithms and gives you some examples of how you can implement these in C++. Since it is important to optimize the code, the book also leaves you with some tips to help you do the same. The book also provides some information on how you can test the code you have written and debug the errors. There are numerous programs and examples given in this book to help you understand how to implement various concepts of C++. Best of luck to you. I hope you enjoy your journey.

References

10 Tips for C and C++ Performance Improvement Code Optimization. (n.d.). www.thegeekstuff.com website: https://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/

C++ Tutorial - Tutorialspoint. (2019). Tutorialspoint.com website: https://www.tutorialspoint.com/cplusplus/index.htm

C++ Programming Language - GeeksforGeeks. (2012). GeeksforGeeks website: https://www.geeksforgeeks.org/c-plus-plus/

Debugging and Testing. (2020). Umn.edu website: https://www.d.umn.edu/~gshute/softeng/testing.html