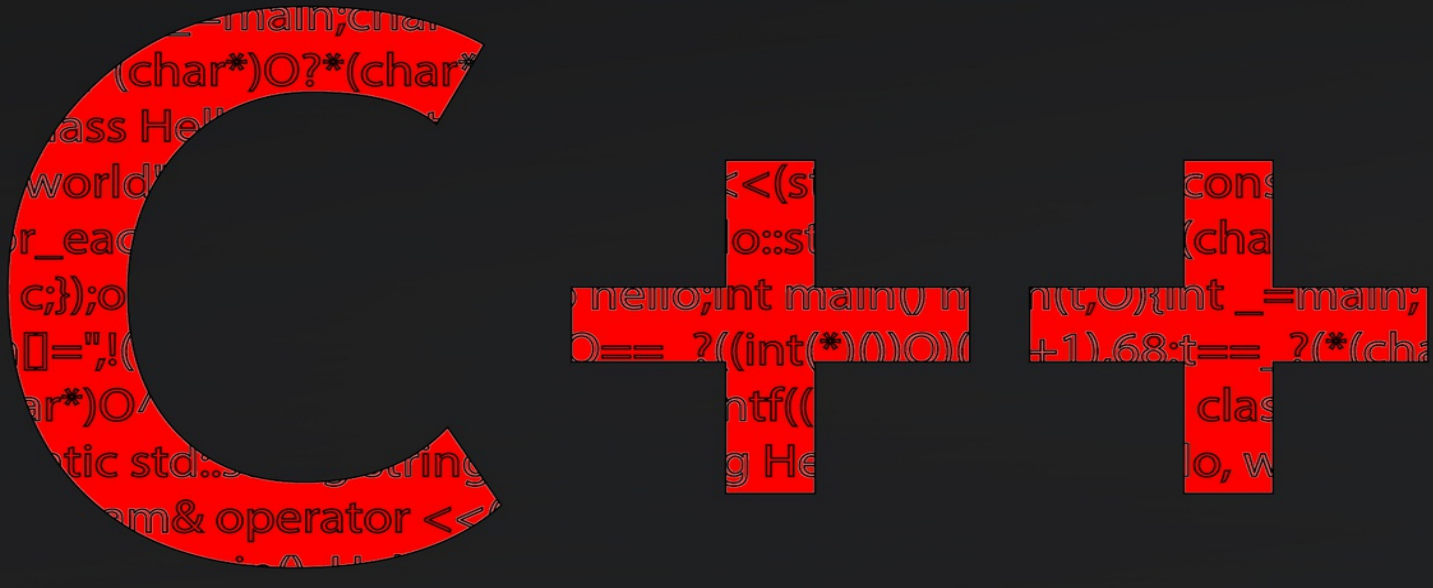


SIMPLE AND EFFECTIVE TIPS AND TRICKS TO LEARN C++

PROGRAMMING EFFECTIVELY

BENJAMIN SMITH

C++

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

Table Of Contents

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.

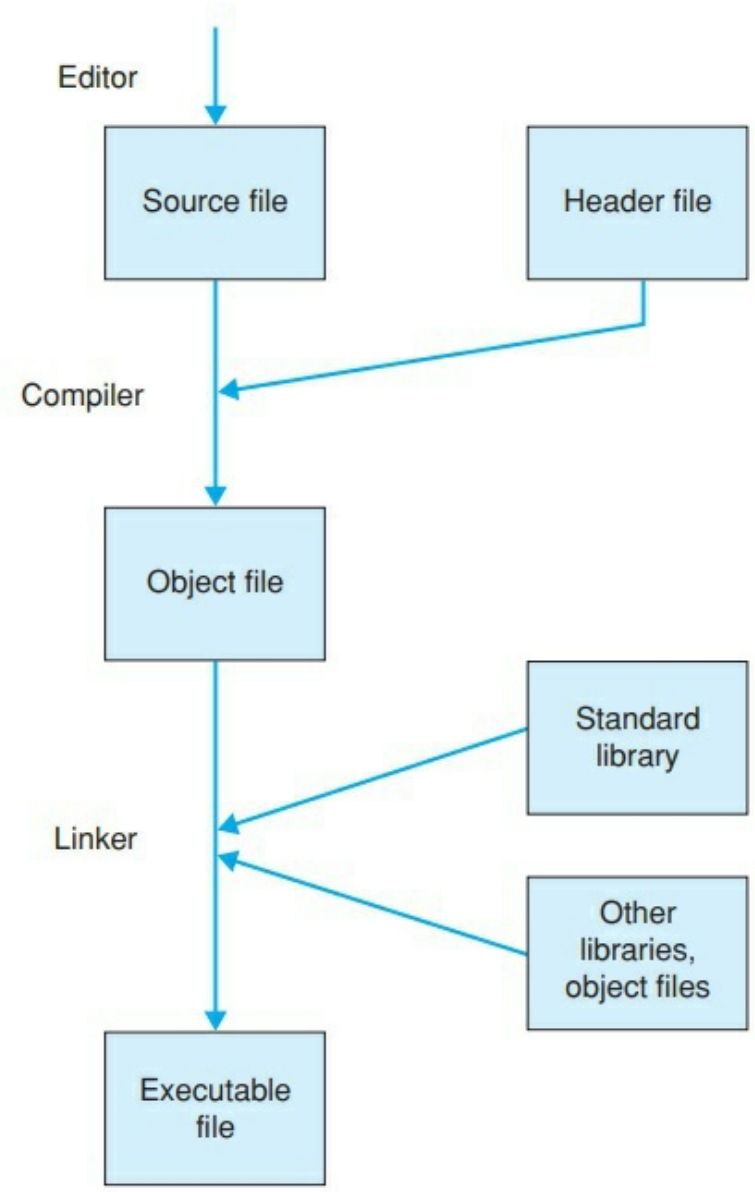
- **Ease of reuse:** objects that have been created are capable of maintaining themselves. Due to this characteristic of objects, this makes them easier to use in other programs as building blocks very conveniently.
- **Low maintenance needs:** According to the situation, an object is capable of modifying 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.

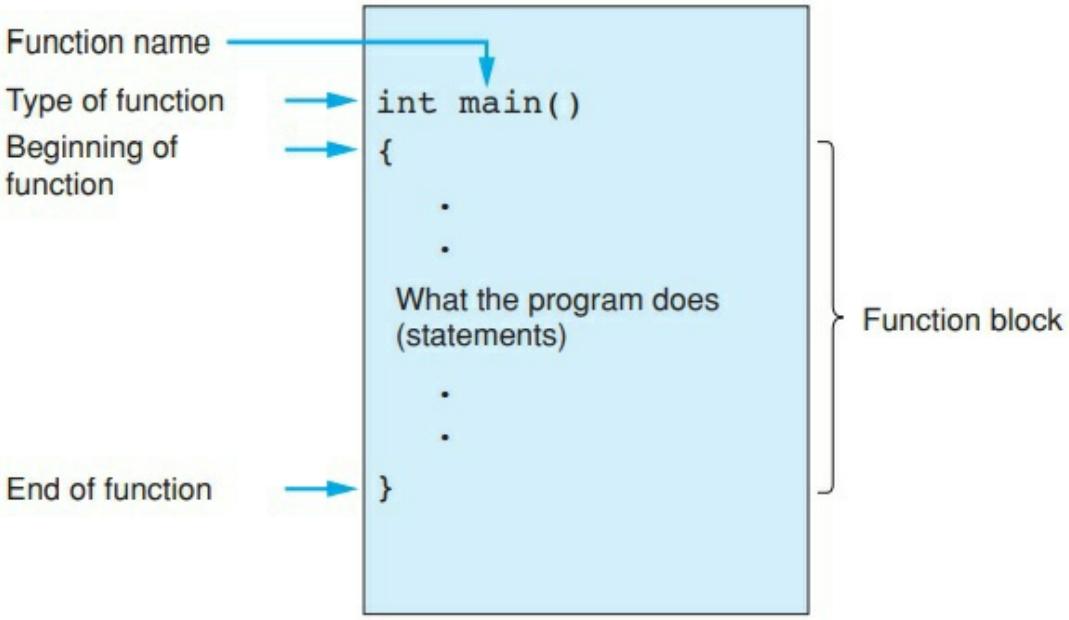
Here's the code for the C++ program:

```
#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

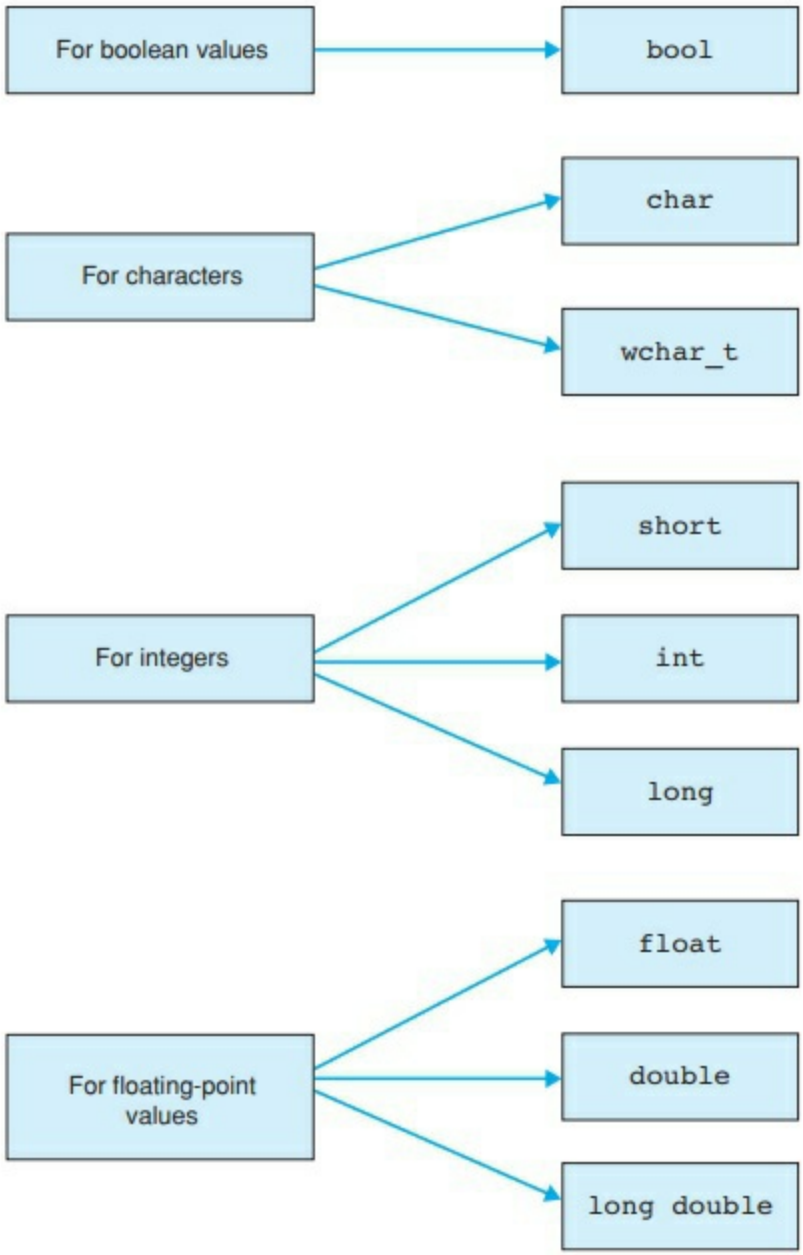
Each data type requires a different approach to be adopted to make it possible for the computer to process the information they carry. That's why it is very important to identify the data type in the program. The purpose of a data type is to elaborate:

- 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.

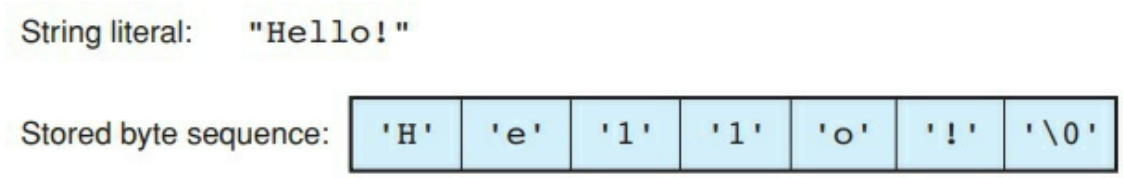
Character Constant	**Character**	**ASCII Code Decimal Value**
'A'	Capital Alphabet A	65
'a'	Small Alphabet a	97
' '	Blank space	32
'.'	Dot	46

‘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	the numerical value of	hh (hexadecimal)

digits)	a character	

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:

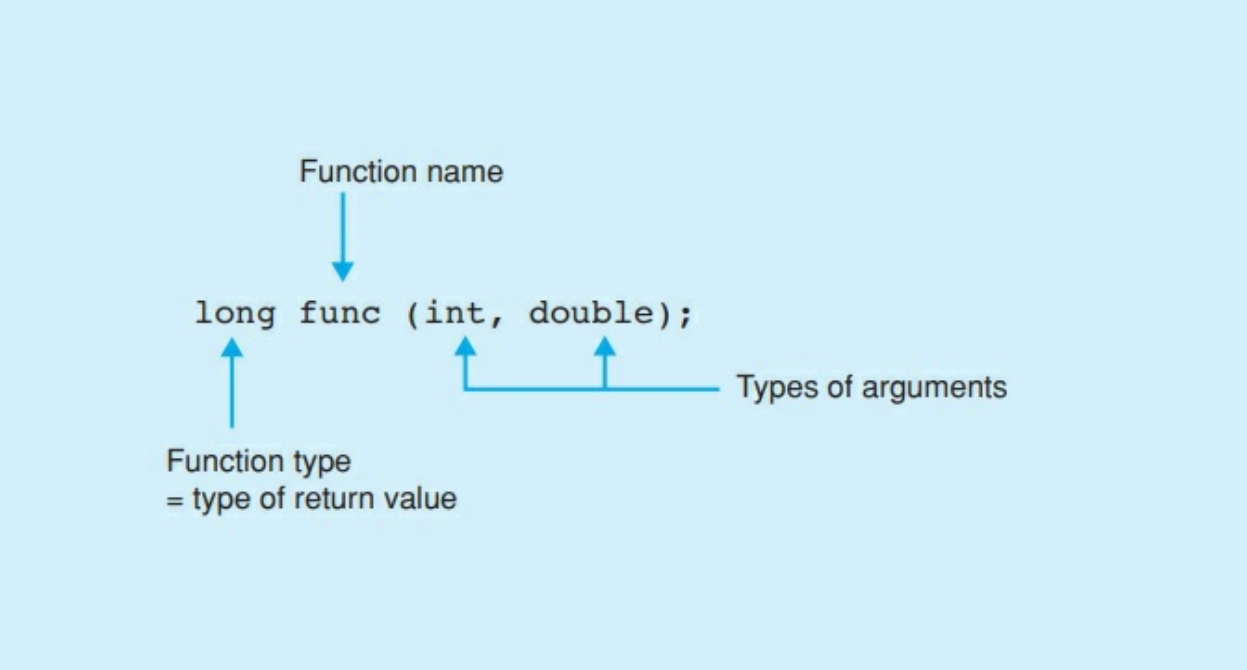
```
volatile const unsigned time_to_live;
```

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.

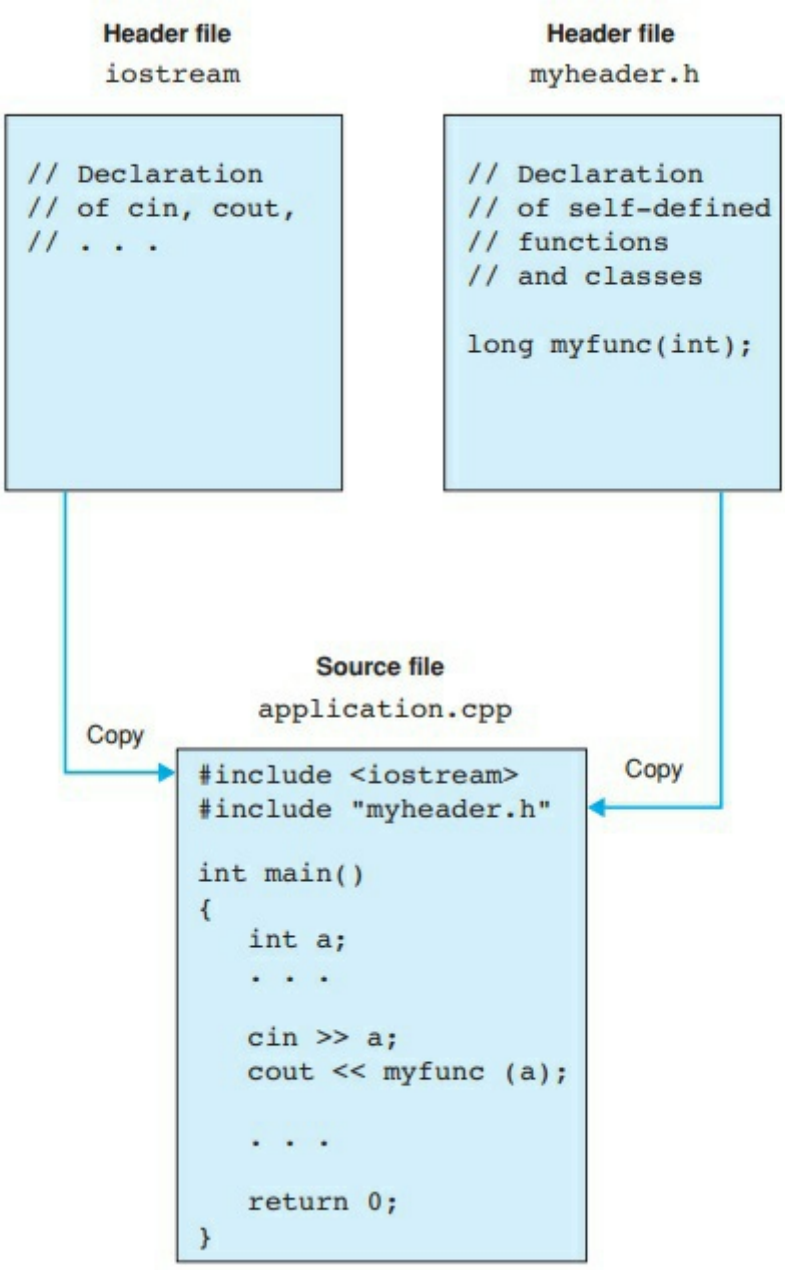
Now that we have discussed a function that can perform an action without returning a value let's discuss a function that does not expect an argument. If we have defined a function without specifying any arguments, then the programmer must declare this prototype function as **void**. Alternatively, we 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:



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.

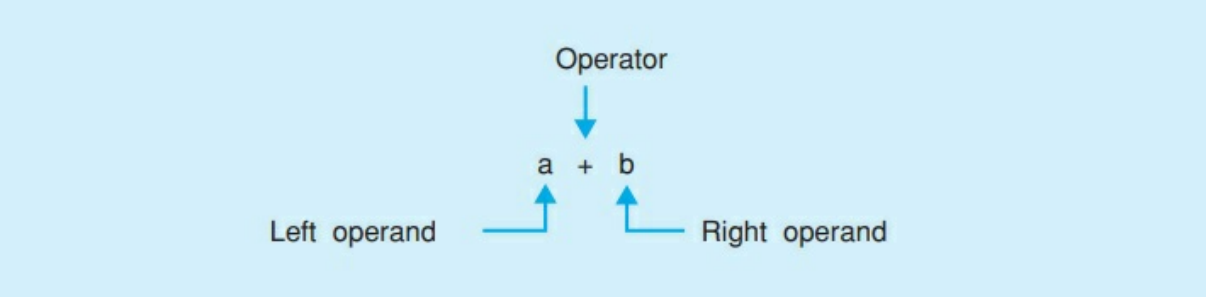
Chapter 4: Operators For Fundamental Types

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 ↕ Low	++ -- (postfix)	left to right
	++ -- (prefix) + - (sign)	right to left
	* / %	left to right
	+ (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

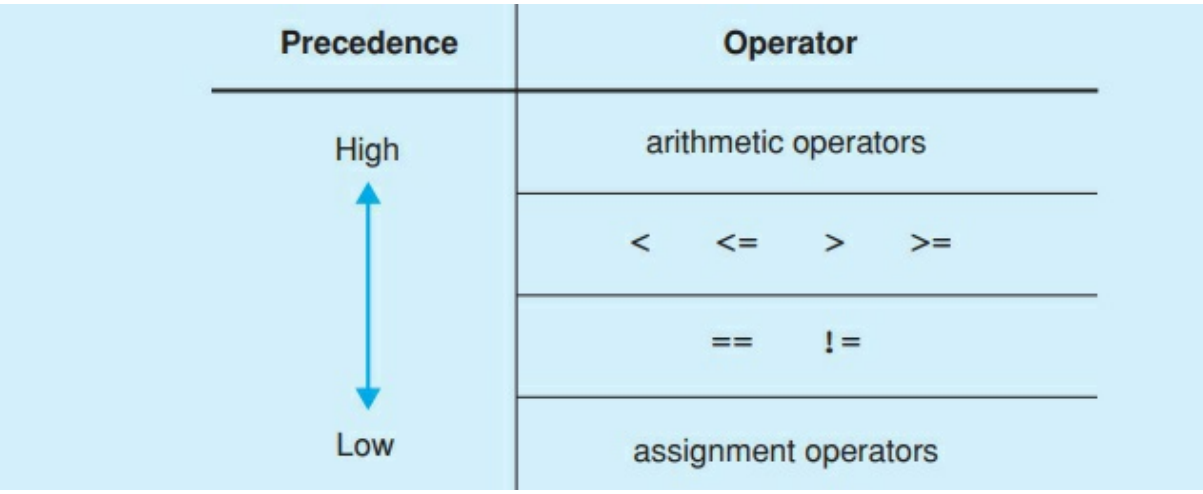
that the program readability is not impaired.

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.

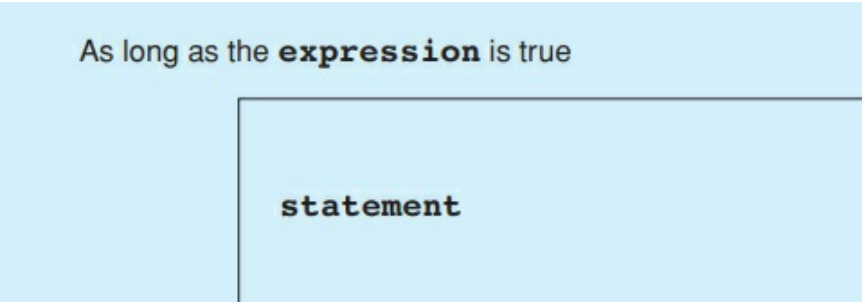
The NOT operator ‘!’, being the unary operator, among others, has higher precedence in the evaluation order.

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.

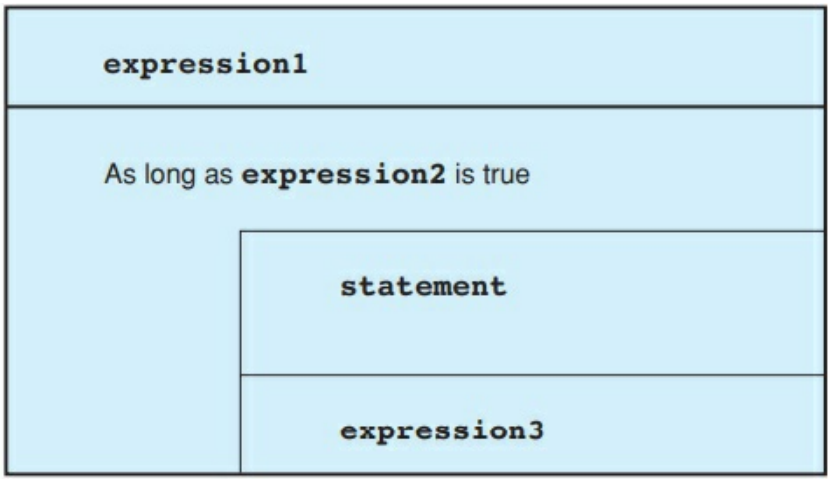
In the following sample program, the average of a sequence of integers is calculated. The loop body contains two statements to be repeated in the process. Hence, both of these statements are placed in a block marked by parentheses.

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.

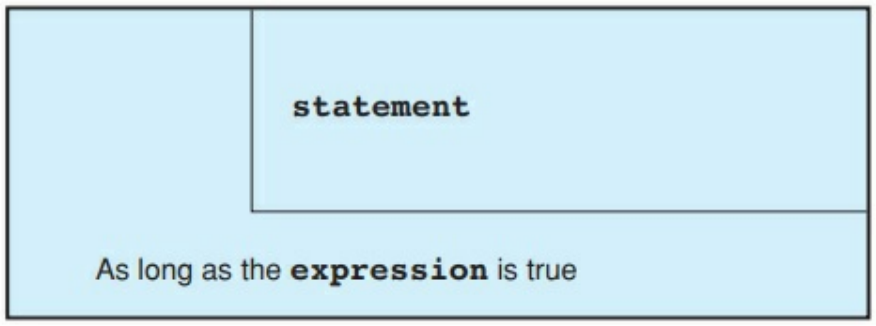
The last expression in a statement containing commas determines the type and value of a comma operator.

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.

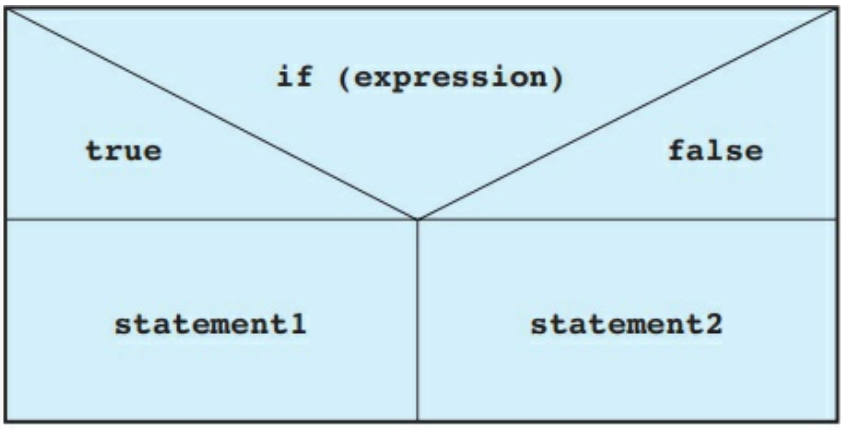
The sample program, as shown, gives an output of several tones as determined by the user input.

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

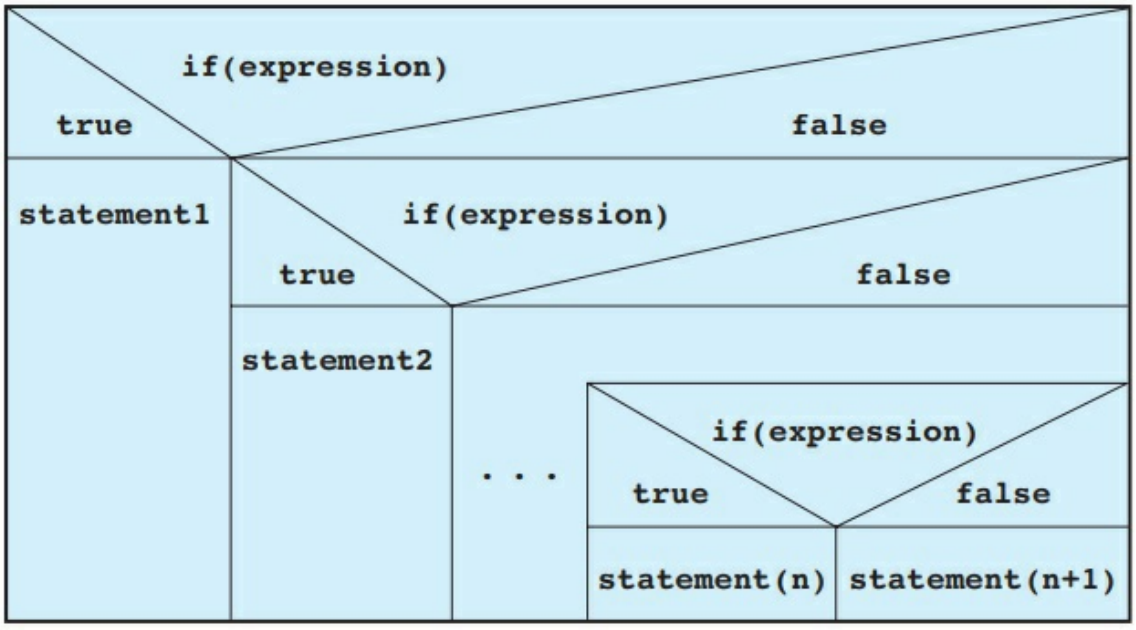
A variable can be defined within the if statement and used for initialization. If the variable is converted to a bool type and returns the value true, then the expression in if statement will also be true.

```
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.

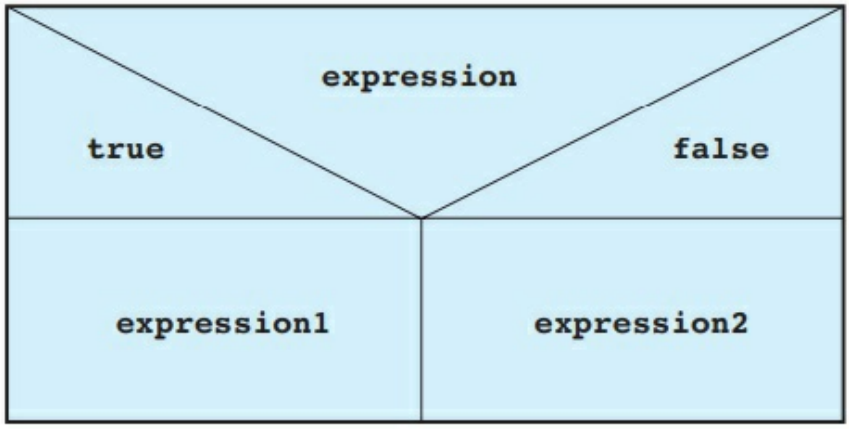
The Sample Program

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

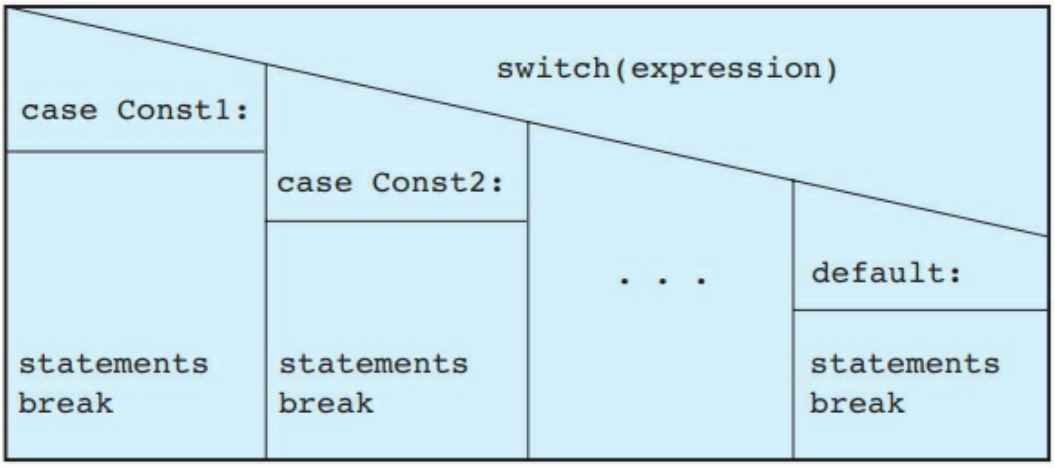
The only operator that uses three operands in the C++ language is the conditional operator. The brackets visible in the first example can be removed because the precedence of a conditional operator is higher than assignment operators and comma operators and lower than the rest.

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

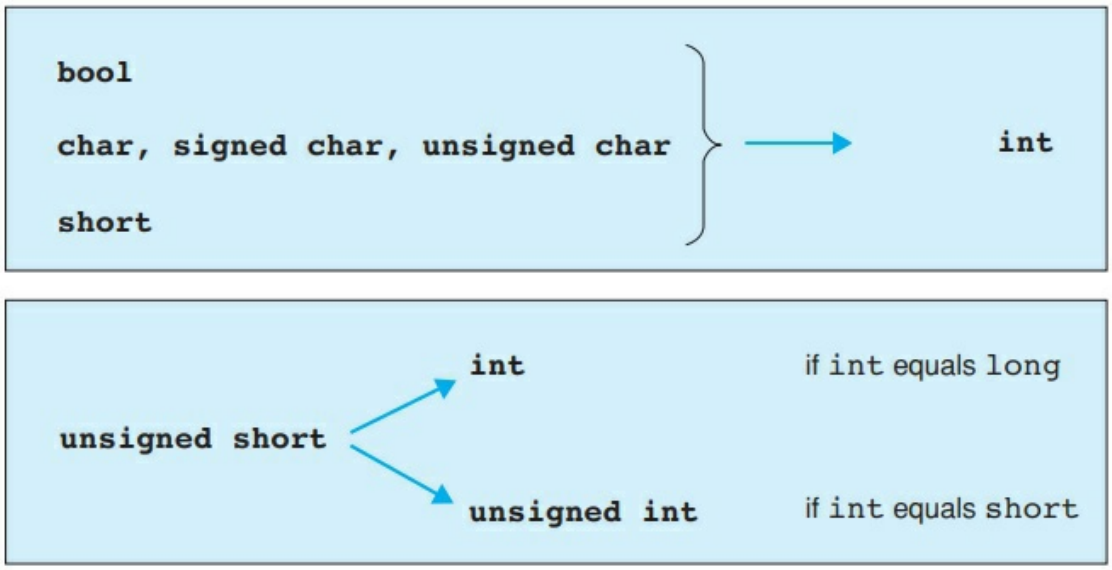
In programming, it is very common to have arithmetic types which are different from each other. Just as the types are different, it's a given that we cannot perform mathematical operations on inputs belonging to two different arithmetic types. When dealing with such types, we need to convert one of them 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.

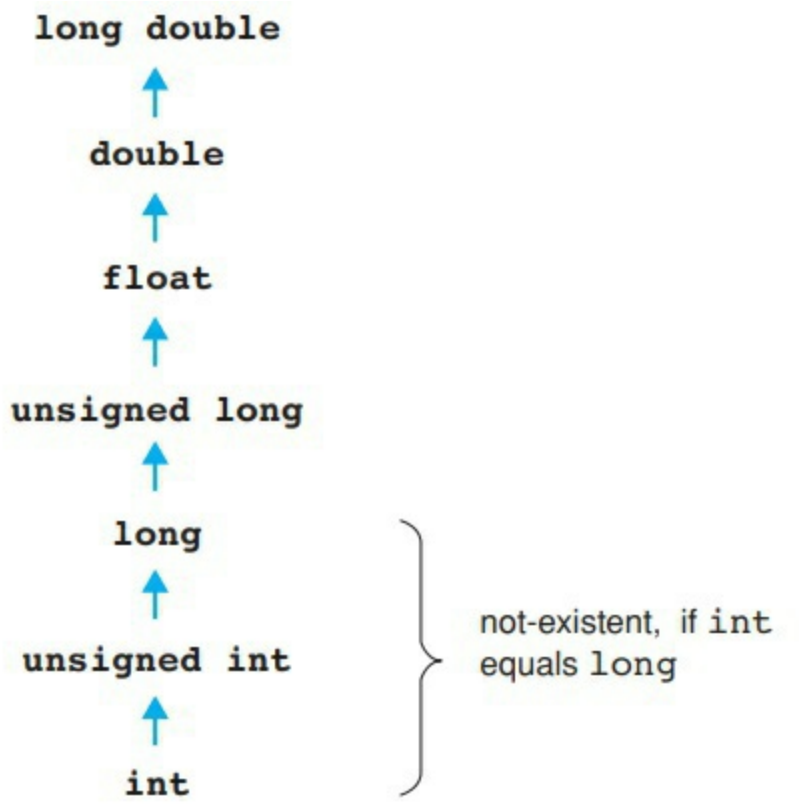
In this chapter, we will chiefly learn about **implicit type conversions** and briefly discuss some additional type conversions at the end of this chapter.

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.

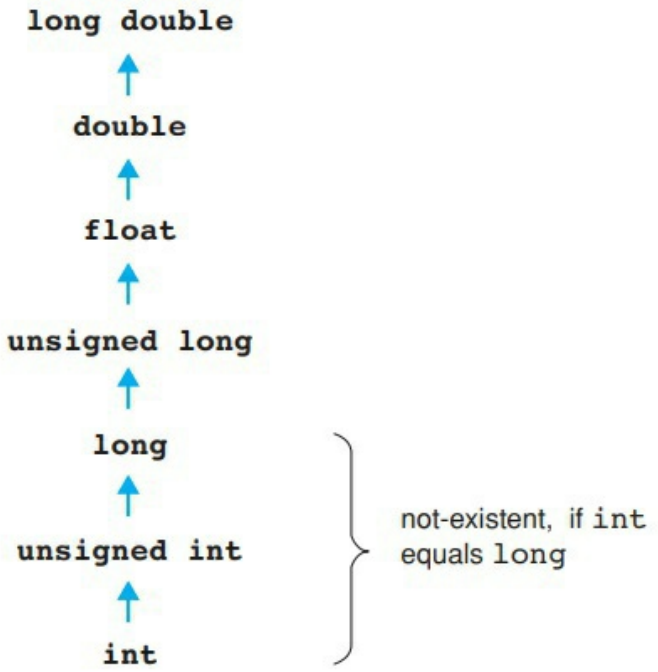
- If an expression is an **unsigned short** type, then it is converted to the **int** type only if it is bigger than the **unsigned short**. If the **int** is not greater than the **unsigned short** type, then it will be converted to **unsigned int** as well as in other cases.

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
```

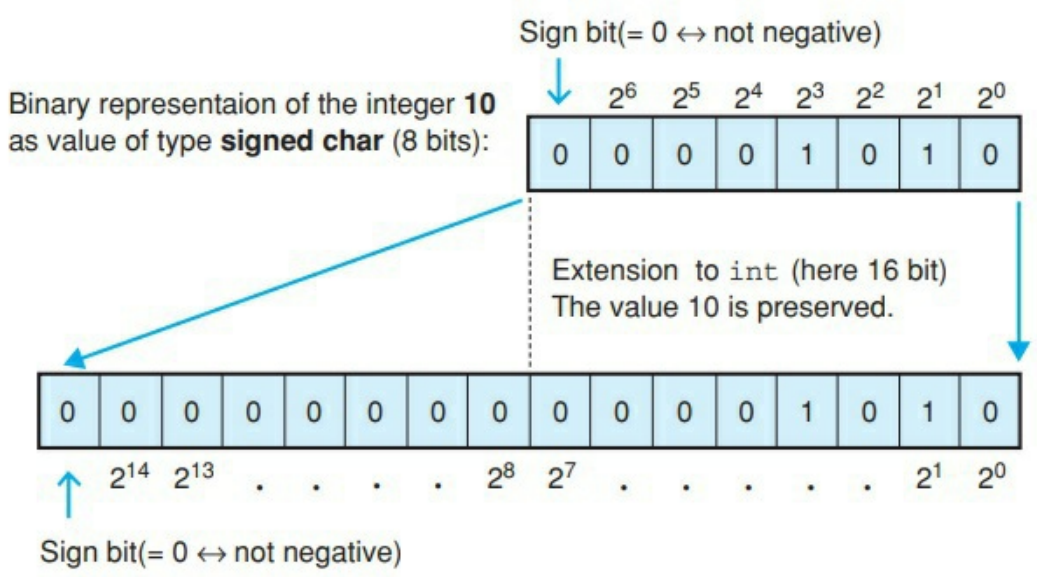
In this example, we are performing two mathematical operations, i.e., division, and multiplication. The original type of the **size** variable is **short**. Before we can perform the division operation in **size/10**, the type of the **size** variable is promoted from **short** to **int**. Once we obtain 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.

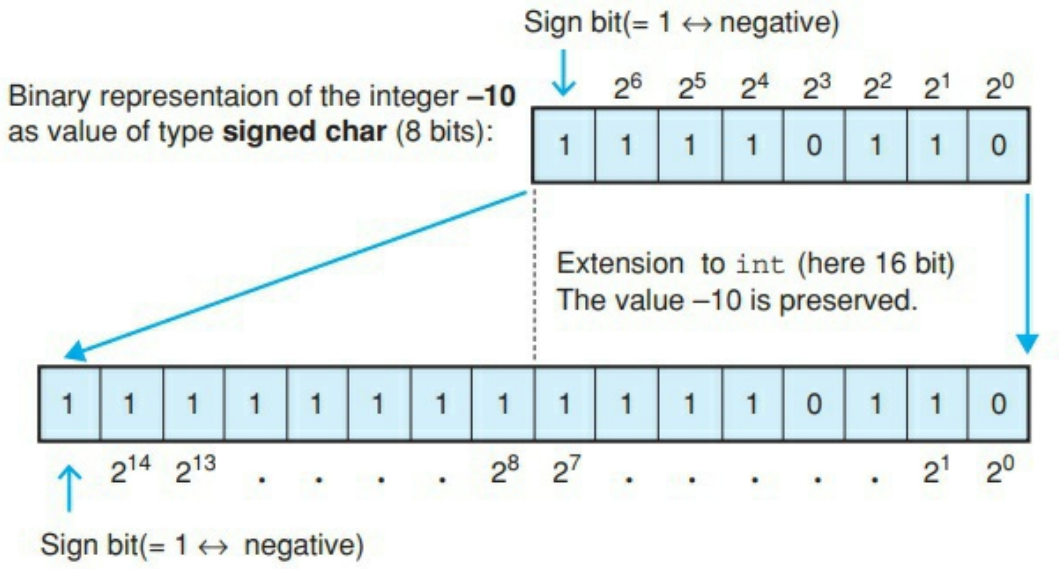
Now let's discuss a few cases where we can apply the Usual Arithmetic Type Conversions.

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.



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

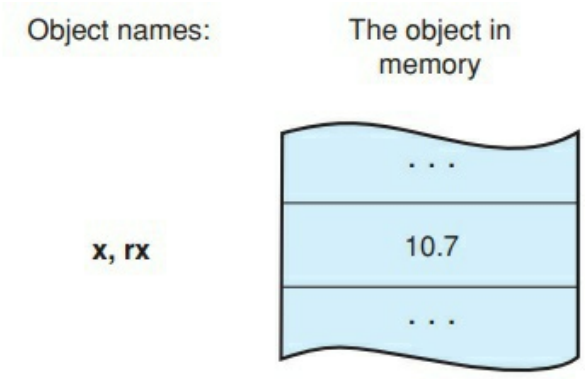
A variable or object that already exists in a specific location in the memory is given another name or alias, which is its reference. This variable can thus be accessed by using its original name or reference. Defining a reference for an object does not mean that it'll occupy extra memory in the program. 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

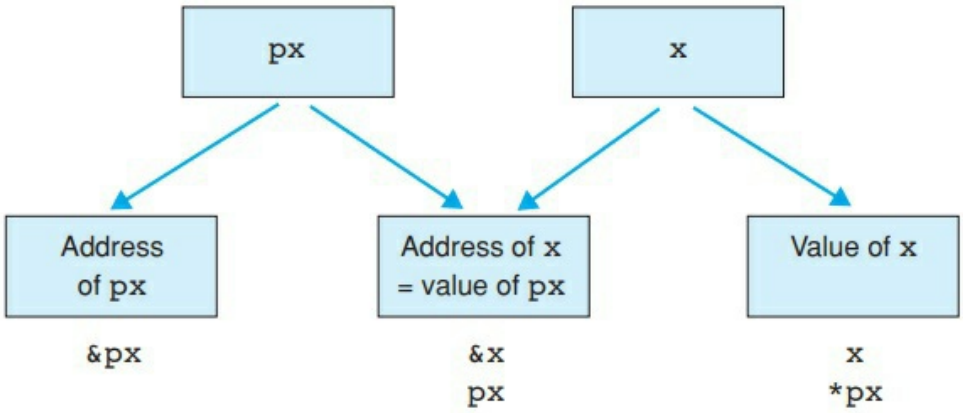
The similarity between references and pointers is that both refer to an object that is stored in the memory. Pointers differ in the aspect that they are not just an alias set for the object they reference but are individual objects with an identity of their own. Although the address for the pointer is 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.

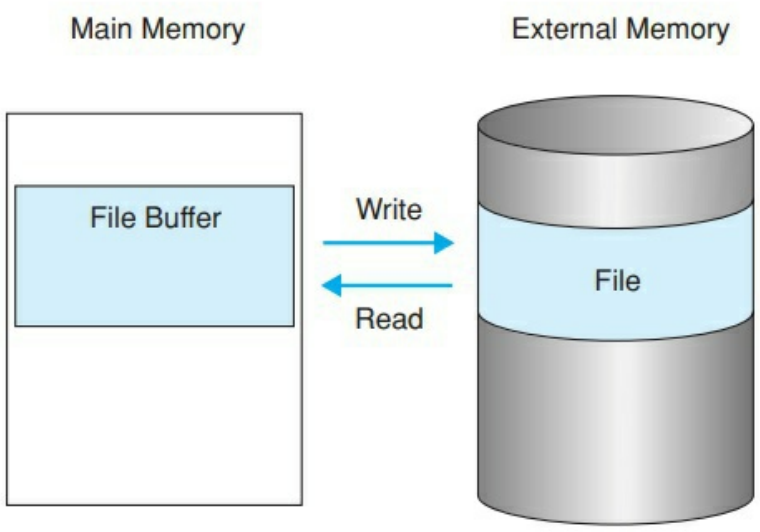
File Operations

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.

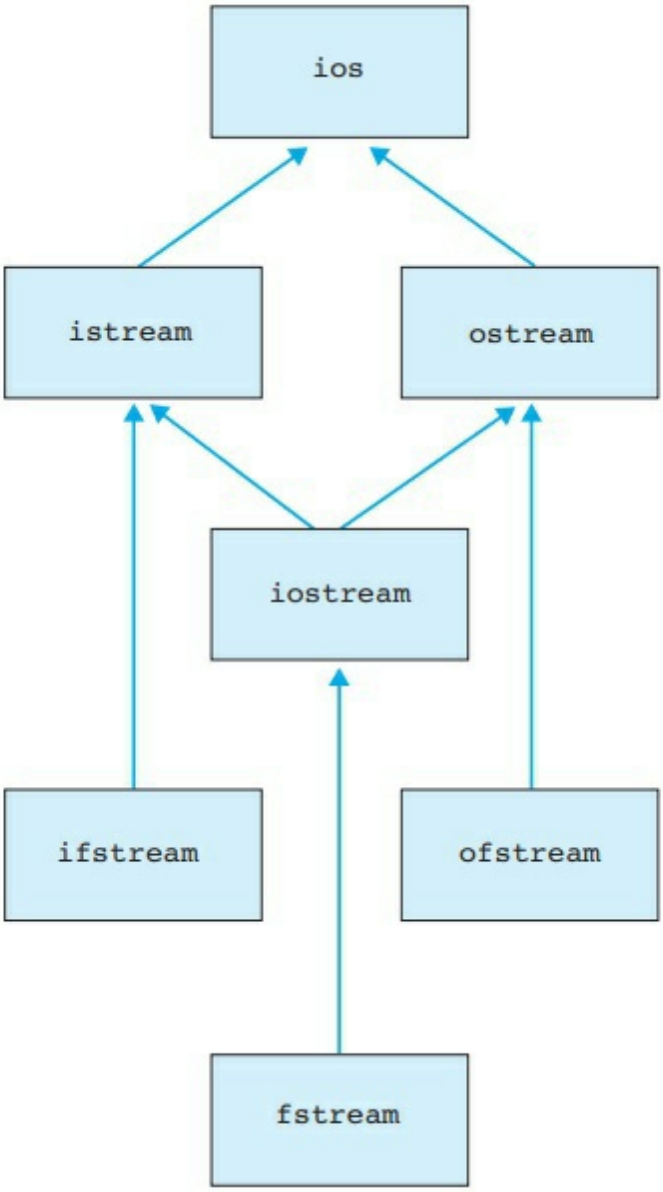
In a file, there are many characters represented by bytes. Each byte in a file holds a specific position. For instance, the first byte of the file will be assigned the position 0, the next byte will be assigned the position 1 and this trend continues. As a consequence of this feature of files, a very 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.**