# C++ Programming Fundamentals

D. Malhotra | N. Malhotra

# C++ Programming Fundamentals

# C++ Programming Fundamentals

Dheeraj Malhotra, PhD
Neha Malhotra, PhD

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

D. Malhotra and N. Malhotra. *C++ Programming Fundamentals.*
ISBN: 978-168392-976-5

Library of Congress Control Number: 2022950638

222324321 Printed on acid-free paper in the United States of America.

*Dedicated to our loving parents*
*and beloved students*

# CONTENTS

# PREFACE

The objective of this text is to emphasize the fundamentals of Object-Oriented Programming (OOP) as an introductory subject. It is designed for beginners who would like to learn the basics of the C++ programming language. With this focus in mind, we present various OOP fundamentals, well supported with real-world analogies to enable a quick understanding in order to solve specific, practical problems. This book will serve the purpose of a text/reference book and will be of immense help especially to undergraduate or graduate students of various courses in information technology, engineering, computer applications, and information sciences.

## Key Features

- *Practical Applications:* Real world analogies as practical applications are given throughout the text to quickly grasp and connect the fundamentals of C++ programming. This approach, in turn, will assist the reader in developing the capability to identify the practical problems that could be efficiently solved through an OOP approach.
- *Programs and Output Snippets:* To better understand the OOP fundamentals at a generic level-followed by their implementation using C++ Programming, detailed implementation codes are discussed to elaborate real world applications  along with their output snippets throughout the book. This presentation will assist the readers in easily understanding the subject at conceptual level and their corresponding implementation.
- *Multiple Choice Questions:* To assist students for placement-oriented exams in various IT fields, several exercises are suitably chosen and are given in an MCQ format.

# ACKNOWLEDGMENTS

We are indeed grateful to Chairman VIPS- Dr. S.C. Vats, Vice Chairman VIPS- Sh. Suneet Vats, Director General VIPS- Dr. Ashwani Kumar Sharma, and Dean VSIT- Prof. Supriya Madan of our employer institute, i.e., Vivekananda Institute of Professional Studies (GGS IP University). They are always a source of inspiration for us, and we feel honored because of their faith in us.

We also take this opportunity to extend our gratitude to our mentors: Dr. O.P. Rishi (University of Kota), Dr. Sushil Chandra (DRDO, GOI), and Dr. Udyan Ghose (GGS IP University) for their motivation to execute this project.

We are profoundly thankful to Ms. Stuti Suthar (SAP Labs), Mr. Sahil Pathak (TMB), Mr. Deepanshu Gupta (Tech Mahindra Ltd.), Ms. Aditi Vats (VIPS, GGSIPU) for helping us in proofreading and compiling the codes in this manuscript.

It is not possible to complete a book without the support of a publisher. We are thankful to David Pallai and Jennifer Blaney of MERCURY LEARNING AND INFORMATION for their enthusiastic involvement throughout the tenure of this project.

Our heartfelt regards to our parents, siblings and family members who cheered us in good times and encouraged us in bad times.

Lastly, we have always felt inspired by our readers especially in USA, Canada, and India. Their utmost love and positive feedback for our four authored titles of *Data Structures using C/ C++/Java/Python*, published with MLI, helped us to further improve the current title.

**Dr. Dheeraj Malhotra**
**Dr. Neha Malhotra**
**January 2023**

CHAPTER 1

# C++ AND BEYOND

## Introduction

In this chapter, you will become familiar with the basics of C++, a powerful programming language used by developers for its easy syntax and concepts. Many video games, embedded systems, IoT devices, and resource-heavy AI applications make use of C++. C++ is a general purpose and cross-platform language that offers a vast collection of libraries to help you program with ease. Let's get started.

## 1.1 The Origin of C++

In 1980, Bjarne Stroustrup, a Danish Computer Scientist working at AT&T Bell labs in Murray Hill, New Jersey, developed C++, which inherited the features of C and Simula67 and supported Object Oriented Programming (OOP). The programming language was originally named "C with Classes," as Stroustrup's main objective was to make use of classes to implement OOP features.

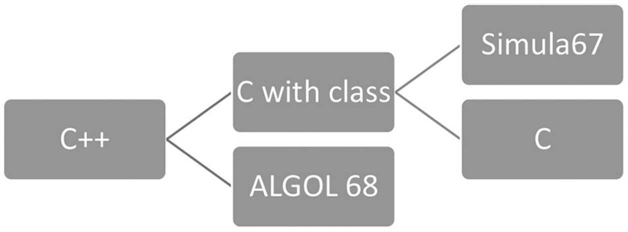


***Figure 1.1*** Programming relationships with C++

The ++ symbol with C is called the *increment operator*, which signifies that the language is an extended version of C. C++ was available outside Bell Laboratories in 1985, and its first C++ compiler, called Cfront, was released in 1985. The American National Standard Institute (ANSI) formed a committee for C++ in 1989. The first draft standards were published in 1995. Programmers are frequently advised to learn C before diving into C+, but that is unnecessary for using this book. We will learn the core principles of C++, as well as some programming basics. C++17 is the latest version of C++ being used.

## 1.2 Why Use C++?

C++ is often used for its simplicity and OOP facility. Other reasons to use this language are as follows:

**C++ Features**

- Simple syntax: As it is derived from C, the syntax is similar and easy to understand.
- Object-oriented: OOP features like polymorphism, inheritance, encapsulation, and abstraction are used.
- Platform dependent: Your code will execute only on the operating system it is designed and developed on.
- Provides memory management: Dynamic memory allocation is supported by C++.
- Mid-level programming: It is capable of doing low-level programming tasks, such as for drivers and kernels, and high-level applications like games, GUI, and desktop apps.
- Vast libraries: You have access to many built-in functions (libraries), which saves time.
- Compiler-based: Your C++ program won't be executed without passing the compilation barrier.
- Structural programming language: Your code is *modular*, i.e., consisting of functions, classes, and objects.

## 1.3 Various Programming Paradigms

### 1.3.1 Structural Programming

*Structural programming* is a programming paradigm in which the entire program is broken down into sub-procedures and functions to make it easier to manage. For example, let's suppose you need to build a simple calculator in C++. You might break it into smaller units that perform different functionalities like addition and subtraction. This approach can be seen in daily life, too: we often take large tasks and break them down into smaller ones that are easier to accomplish. Examples of structural programming languages are C, C++, Java, and C#.

The characteristics of structural programming languages are as follows:

- Modular programming
- Data and functions treated as separate entities
- Program designed using top-down approach
- Procedure (algorithms) gets more attention than the actual data used
- Larger programs are divided into smaller units called *functions*.

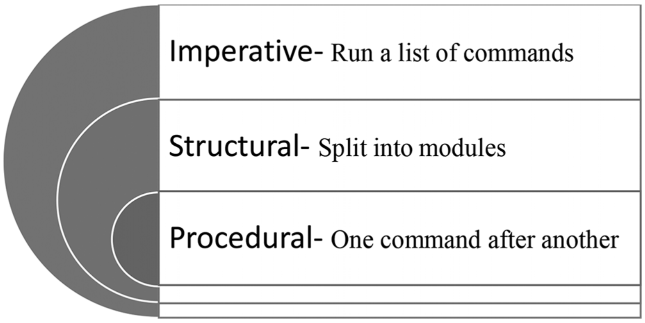

Imperative- Run a list of commands
Structural- Split into modules
Procedural- One command after another

***Figure 1.2*** Types of paradigms

Consider the following example. The steps needed to make a cup of instant coffee are as follows:

Step 1. Check for the availability of coffee.

Step 2. If true, find sugar and water or else go back.

Step 3. If true, then mix all three or else go back or leave the coffee sugar free.

Step 4. Check for the availability of milk.

Step 5. If true, add milk to black coffee

Step 6. Stop

The above example shows how an algorithm to make coffee follows steps with decision statements and creates a kind of structure.

Can you think of the advantages and disadvantages of following this approach? Let us list them.

***Table 1.1*** Structural programming Advantages & Disadvantages

| **Advantages** | **Disadvantages** |
|---|---|
| Easy to understand and user friendly | Gives more attention to procedure/code |
| Easy to find errors | Data and function treated as separate entities |
| Machine independent | Time-consuming have to convert to machine code |
| Maintenance is easier as less complex | Sensitive data is not safe |
| Works from the top, down | Top-down approach is not applicable when programs are large |

### 1.3.2 Procedural Programming

*Procedural programming* comes under the umbrella of structural programming, but the program executes in a particular order, with one statement after another. It consists of procedure calls, and for a particular block of code of a procedure, no special flow-alter statements are used. Examples of such programming languages are Pascal, COBAL, C, and Fortran.

### 1.3.3 Object Oriented Programming

Programming data has set limits, unlike the myriad amounts of data that are generated by us on a daily basis. Broadly speaking, a *class* encompasses methods and data grouped together as members (i.e., they are *encapsulated*). Hence, objects in C++ store data and have a defined state and behavior (also known as an *instance of a class*).

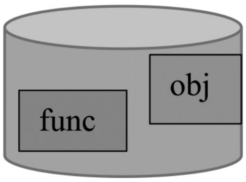


***Figure 1.3*** Encapsulation

#### 1. Encapsulation

Shielding data from any threat involves binding it together with methods in objects, and so whenever we need to call upon any functionalities, our data is provided along with them. Look at it like a capsule with all the important ingredients you might need all in one place. C++ implements this concept by introducing classes into our program.

### 2. Data Abstraction

*Abstraction* involves fetching only the relevant data to the user and hiding any other details. Consider an example where you go to a restaurant and place your order. You sometimes get an order number to pick up your food when the number is called. You are not concerned with the tools or number of employees working to make your meal or their details. Relevant information sharing saves time and memory, and this protects our data.

We can implement abstraction in C++ using classes to group data members and member functions together. This approach will determine which data member will be accessible to which extent to other entities in our program.

### 3. Polymorphism

*Polymorphism* provides us with multiple forms of a method with different signatures but the same name. There are two types: runtime polymorphism and compile-time polymorphism, both of which are implemented in C++ using method overloading and over-riding. Methods can exist in multiple forms by varying the type of parameters and number of parameters they take in the signature of the function. When these methods used with same name but different signature it is called overloading and when used with same name as well as signature is called over-riding.

Let's consider the following example:

In Formula 1 Racing, there are 10 teams with 2 drivers each, and the drivers are provided with the same equipment and car specifications. However, the result attained by both drivers for their teams always differs as the "method" (the car) has been "overloaded" in completely different ways (that is, the F1 car exists in different forms).

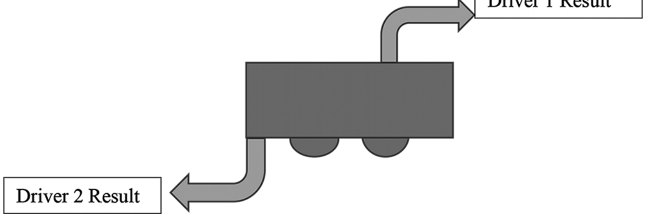


***Figure 1.4*** Polymorphism

### 4. Inheritance

Just as all children inherit certain qualities from their parents and grandparents, classes in C++ can also inherit certain properties or attributes from other classes. All you need to learn at this stage is that the class that inherits from the parent is called the *derived class*, while the parent class is called the *base class*. This promotes the reusability of the code blocks, and you might not need to define all classes.

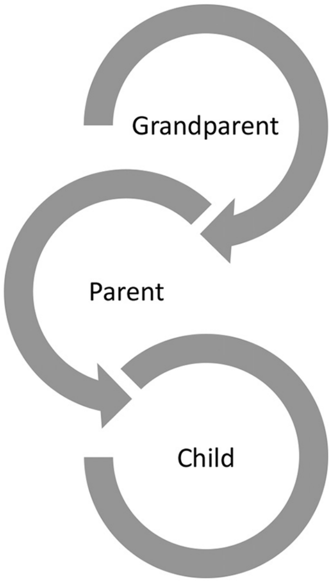


***Figure 1.5*** Inheritance

Let's consider an example:

As seen in many industries, nepotism plays a significant role for younger generations. Let us say that an actor has worked hard to become famous. This reputation would be passed on to his children when they to enter the world of cinema. They will do this with considerable ease using their father's inherited reputation. In this case, the actor is the base class and his children are all derived classes.

### 5. Message Passing

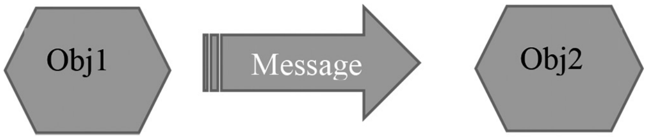


***Figure 1.6*** Message passing

Communication is important for objects in C++, as they might need to execute some requests from other objects to complete their own task (function). For successful communication, you need the sender's and receiver's address and information to be sent. For C++ terms, all you require is the correct object name(receiver), function name, and data.

**6. Dynamic Binding**

*Dynamic binding* (or *late binding*) tells which procedure will be called at runtime. The code we write in C++ is called the *source code* and it is saved with the extension .cpp. The compiler checks each line in the program to find any errors and notify us about them. The time it takes the program to execute is called the *runtime*. That is when the function calls are executed. The code inside our procedure is not known until execution, hence the name late binding. In C++, it is implemented using objects.

## 1.4 C++ Basics

Now that you understand what can be done in C++, let us look at exactly how it is done. Similar to mathematics, we will learn about what variables are, as well as the symbols we will use to solve problems.

### 1.4.1 Variables

*Variables* in C++ are unique names given to units that hold value within a defined scope. Naming is done by following certain rules. By following these rules, you can make your program more understandable to others (as well to yourself).

The *variable declaration* means introducing the variable to the program before it is used anywhere.

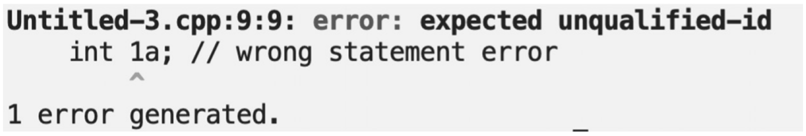
A *variable definition* means the variable is assigned a memory location and a value. Rules for naming variables in C++ are as follows:

1. It should start with an alphabet
2. It can contain combinations of digits or , letters.
3. It should not contain any whitespaces or special characters (such as !, %, or #) except underscore(_).
4. It should not contain any C++ reserve words (also called *keywords*).

**Code: Variable Naming in C++**

```
#include <iostream>
using namespace std;
int main()
{

   //variables naming
    int a,b; // declared
    char ch;
    int 1a; // wrong statement error
     char _abc;// declared and defined
    cout<<"They take value "<<a<<endl;
    return 0;
}
```

**Output:**



```
Untitled-3.cpp:9:9: error: expected unqualified-id
    int 1a; // wrong statement error
        ^
1 error generated.
```

### 1.4.2 Data Types

When a variable name is assigned, the variable will hold value, and this value will be one of the types available in C++.

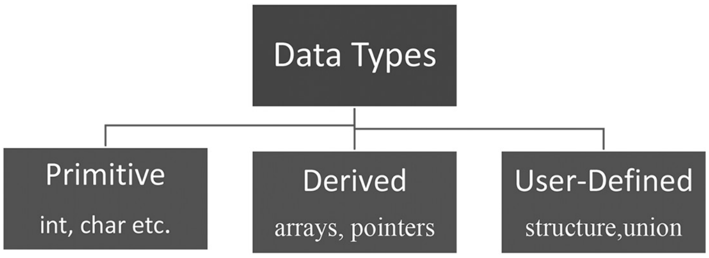


***Figure 1.7*** Data types

#### 1. Primitive Data Type

These can be used directly to declare your variables.

- *Integer*: denoted by `int` and used when you need numeric values. It ranges from -2147483648 to 2147483647 and takes up 2/4 bytes. One byte equals 8 bits, and 2^32 gives its range of memory, depending on the compiler.

- *Character*: used for alphabetic values. The memory space is only 1 byte. It ranges from -128 to 127 or 0-255 for signed and unsigned char data types.
- *Boolean*: Some statements require a logical answer, i.e., either true or false. This is done in C++ using `bool`, which stores a Boolean value.
- *Floating Point*: This data type is used for storing decimal values, such as 1.2 or 1.234, up to 7 digits of precision. The keyword `float` is used for this and takes up 4 bytes of memory space.
- *Double Floating Point*: This data type is also to store decimal values, but with 15 digits of precision. The keyword used for the double floating-point data type is `double`. Double variables typically require double the space of floating point variables, i.e., 8 bytes.
- *Void*: "Void" means nothing, containing no value. However, you should be careful not to confuse it with zero: 0 is a numeric value, while void is valueless. It is mostly used with a function return type to signify that it returns no value.

#### 2. Derived Data Type

This type comprises of pointers and references.

#### 3. User-defined Data Type

You can also create your own data type depending on what the program requires. This type encompasses classes, structures, `enum`, and typedef.

### 1.4.3 Data Modifiers

The above datatypes can be used directly but within their respective scope. *Modifiers* are used to extend the limit of a data type to accommodate larger values. The keywords used are as follows:

1. Signed: This is very useful as it stores all positive, negative values, and even zero.
2. Unsigned: This only stores negative values.
3. Short: This is used for small integer values and ranges from –32,767 to +32,767.
4. Long: This ranges from -2147483647 to 2147483647 and is used for larger values.

## Code: Implementing Data Types in C++

```
//Various data types and their sizes
#include <iostream>
using namespace std;
void main()
{
    int x=98;
    short int sh=1;
    long int l= 34566;
    signed int s= -9978;
    unsigned int u= 66;
    char c='A';
    float f=1.98;
    double d=9.6666788;
    bool b= true;
    cout << "Size of int : " << sizeof(int) << " bytes"
        << endl;
    cout << "Size of short int : " << sizeof(short int)
        << " bytes" << endl;
    cout << "Size of long int : " << sizeof(long int)
        << " bytes" << endl;
    cout << "Size of signed int : "
        << sizeof(signed int) << " bytes" << endl;
    cout << "Size of unsigned long int : "
        << sizeof(unsigned int) << " bytes" << endl;
    cout << "Size of char : " << sizeof(char) << " byte"
        << endl;
    cout << "Size of float : " << sizeof(float) << " bytes"
        << endl;
    cout << "Size of double : " << sizeof(double)
        << " bytes" << endl;
// no return statement as void used
}
```

## Output:

```
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed int : 4 bytes
Size of unsigned long int : 4 bytes
Size of char : 1 byte
Size of float : 4 bytes
Size of double : 8 bytes
```

***Table 1.2*** C++ Data Types and Domain Range

| Data Type | Bytes | Range |
|---|---|---|
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| Short int | 2 | 32,768 to 32,767 |
| Long int | 4 | -2,147,483,648 to 2,147,483,647 |
| Unsigned int | 4 | 0 to 4,294,967,295 |
| Signed char | 1 | -128 to 127 |
| Unsigned char | 1 | 0 to 255 |
| float | 4 | ±3.4E-38 and ±3.4E38 |
| double | 8 | ±1.7E-308 and ±1.7E308 |
| Long double | 12 | ±1.7E-308 and ±1.7E308 |

## 1.5 C++ Execution Flow

Let's talk about a few basic terms before we move on:

- *Machine code* are instructions that only the computer understands, written in binary(0,1), and can be directly executed by a CPU.
- *Assembly code* is a low-level language consisting of statements written in English that can be understood by the programmer.
- *Debugging* means finding the faults or errors in your program to fix them so the program executes correctly.

A program in C++ executes in four stages, where different tools help us to make the execution process easier:

1. *Pre-processor*

   The first event that occurs in a C++ program is that the pre-processor processes the program before compilation. It includes the header files and any macros, if defined. It prepares the source code for the later stages of execution.

2. *Compiler*

   Now the pre-processed source code passes through the compiler to produce an object file. The complier translates the programming language code in C++ into the machine language, and checks for any errors, issuing warnings about the errors. The object file contains the assembly code.

3. *Linker*

   The linker connects the libraries and object files together to generate an executable file with the extension .exe.

4. *Loader*
   The loader places your program onto the memory and this is where the programs actually runs.

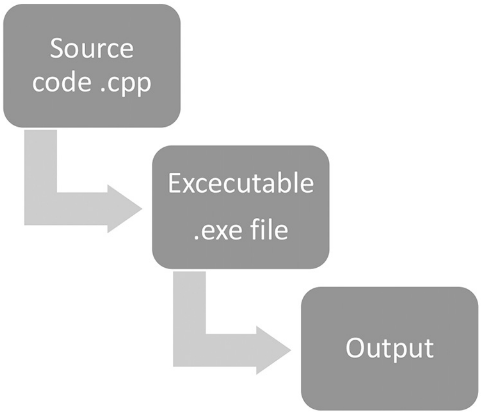


***Figure 1.8*** C++ execution flow

Let us write our first program in C++.

Steps to follow:

1. Get a text editor (such as Notepad) to write C++ code and a compiler(GCC) to translate the C++ code into machine code.
2. Get an IDE (Integrated Development Environment) to edit and compile your code (such as Code::Blocks, Eclipse, or, in this example, Visual Studio).
3. In Visual Studio, go to `Folder>File>new.cpp` and save the file before you start coding.
4. Click on the right-most run symbol, which shows you options to debug or run the file to get output on the code terminal.

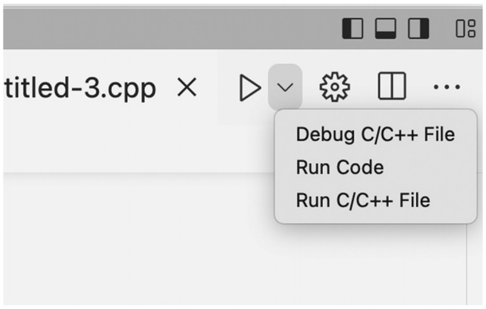


***Figure 1.9*** Visual Studio code options

5. Another way to run code is to type it into Notepad and then save it as a .cpp file. Open your command prompt terminal and type the `run >gcc filename.cpp` command to see output.

### Hello New You Program

### Code: First C++ Program

```
#include <iostream>//pre processor
using namespace std;//libraries used

int main() {//first block that runs
//
  cout << "Hello new you!!";//your first output
  return 0;// as the main method not void
}
```

### Output:

```
Hello new you!!
```

## Summary

- C++ is a compiled, general-purpose, case-sensitive, and free-form programming language that supports procedural, object-oriented programming.
- C++ was developed by Bjarne Stroustrup in 1980 at Bell Labs.
- There are programming paradigms, such as procedural, structural, and object-oriented.
- C++ has OOP features, such as polymorphism, inheritance, data encapsulation, abstraction, message passing, and late binding.
- We learned about C++ data types, data modifiers, and variables.
- We looked at the ranges for `int`, `char`, and `long int`.
- We discussed working with a compiler and debugger.
- We considered the stages of C++ program execution.
- We wrote our first program in C++ using simple steps.

## Exercises

### Theory Questions

1. What are the features of C++? Why do we need to learn C++?
2. List four practical field applications of C++.
3. Differentiate between types of programming paradigms.
4. Explain OOP concepts with valid examples.
5. List the rules for naming a variable. Identify the valid variable names from the following list:
   - **a.** _thisis2
   - **b.** 24Hours
   - **c.** Name@dune
   - **d.** Formula1
   - **e.** __aditi
   - **f.** Meine#Name
6. What are the tools used in C++ program execution?
7. Differentiate between compiler and debugger?
8. How exactly does a C++ program execute?

### MCQ-Based

1. Who is the original creator of the C++ language?
   - **a.** Dennis Ritchie
   - **b.** Ken Thompson
   - **c.** Bjarne Stroustrup
   - **d.** Brian Kernighan
2. C++ is a ___ type of language.
   - **a.** high-level
   - **b.** low-level

**c.** middle-level

**d.** None of the above

**3.** Which of the following symbols can be used to name a valid variable?

**a.** %

**b.** _

**c.** *

**d.** All of the above

**4.** Structure is what kind of data type?

**a.** Built-in

**b.** Derived

**c.** User-defined

**d.** None of the above

**5.** Which of the following features are supported by C++?

**a.** Encapsulation

**b.** Inheritance

**c.** Polymorphism

**d.** All of the above

**6.** What value does void hold?

**a.** Not defined

**b.** Null

**c.** 0

**d.** None of the above

**7.** Inside a class, the declared data members are known as __________

**a.** data

**b.** object and data

**c.** members

**d.** None of the above

8. What is the output of following C++ program?

```
#include <iostream>
using namespace std;
int main ( )
{
   double i;
  i = 15;
  cout << sizeof(i);
 return 0;
}
```

**a.** 2

**b.** 3

**c.** 4

**d.** 8

9. The declaration of a variable means to _________ it?

**a.** introduce

**b.** delete

**c.** give value to

**d.** use

10. Object-oriented programming employs a _________ programming approach.

**a.** top-down

**b.** procedural

**c.** bottom up

**d.** All of the above

## Practical Application

1. Write a program in C++ to check the upper and lower limits of an integer.
2. Write a program in C++ to print your name and details on separate lines.
3. Write a program in C++ to find the size of the following variable: `abc`.

4. Write a program in C++ to find the sizes of various data types and print them.
5. Write a program in C++ to display the use of naming variables and their types.
6. Write a program in C++ to check whether variable `234Mine` exists.

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (c) | **2.** (c) | **3.** (b) | **4.** (c) | **5.** (d) | **6.** (b) | **7.** (c) | **8.** (d) | **9.** (a) | **10.** (c) |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition), (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition), (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++*, (BPB Publications, September, 2020).

### Websites

- Learn CPP, accessed May 2022, *https://www.learncpp.com*
- Silly Codes, accessed May 2022, *https://sillycodes.com*
- Codes Cracker, accessed May 2022, *https://codescracker.com*
- Geeks For Geeks, accessed May 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed May 2022, *https://www.udacity.com*
- Scaler, accessed May 2022, *https://www.scaler.com*
- C Plus Plus, accessed May 2022, *https://cplusplus.com*

CHAPTER 2

# Basic Play in C++

## 2.1 Literals, Constants, and Qualifiers

Often while coding, your program will need a value that will not change throughout the program's life, such as the value of pi or any basic values you wish not to manipulate. A fixed value that may not be changed is called a *constant*. *Literal constants* (or just *literals*) are unnamed values used directly in our code. They are constants because their values cannot be changed until you rewrite the program and compile it again. Just like objects have a type, all literals have a type. The type of a literal is assumed from the value and format of the literal itself. *Character literals* are enclosed in single quotes. A character literal can be a plain character, escape sequence, or universal character. A special character with a \ at its beginning has a specific meaning and performs a certain function.

***Table 2.1*** Escape characters

| Escape Sequence | Meaning |
|---|---|
| `\\` | `\ character` |
| `\'` | `' character` |
| `\"` | `" character` |
| `\?` | `? character` |
| `\a` | `Alert or bell` |
| `\b` | `Backspace` |
| `\f` | `Form feed` |
| `\n` | `Newline` |
| `\r` | `Carriage return` |
| `\t` | `Horizontal tab` |
| `\v` | `Vertical tab` |

A *qualifier* is a keyword that is applied to a type, making it into a qualified type (this is much like using `const int` as a qualified type, which means the value is an integer that will not be changed throughout the program).

**Code: Use of escape Sequences**

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

int main() {
   cout <<  "\nHello\tProgrammer\n\n ";
   return 0;
}
```

**Output:**

```
Hello  Programmer
```

## 2.2 Stream-Based IO

In C++, for the user to enter values, we use input streams; here, we use the term *stream* to refer to a group of bytes that can be accessed sequentially. There are two different types of streams. *Input streams* are used to hold input from a user, such as the information that comes from a keyboard. For example, the user might press keys on the keyboard, and all these keystrokes are saved in the input stream to be used later when the program itself requires it. *Output streams* are used to showcase output using a monitor, a file, or a printer. For example, you may need a printout, but the printer is currently printing another document, so your data/file waits for its turn to be given as output. To able to use all these functionalities in your C++ code, you need to include the *iostream header file*, which provides with a whole hierarchy of classes (multiple inheritance) to make use of all I/O classes.

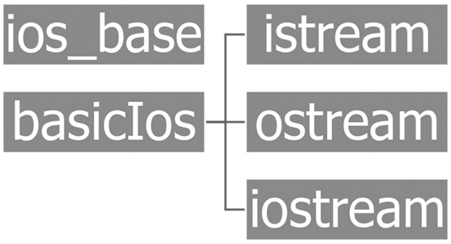


**Figure 2.1** IOStream

In this section, you will work with `iostream`; the `iostream` class is derived from `ios_base`. The symbols << and >> are special operators. The `istream` class is for input streams. The *extraction operator* (>>) removes values from the stream created when the user presses keys. The `ostream` class is for output streams, and the *insertion operator (<<)* is used to put values in the stream to be displayed when asked on output devises like monitors. The `iostream` class can handle both input and output streams, whether it is the user pressing random keys or an alert message to the user not to do so. This is all accomplished through one class.

The *standard stream* is a stream provided to your computer program by its own environment. C++ comes with the following predefined standard stream objects.

1. **cin** is an `istream` class that uses standard input keys for fetching values.
2. **cout** is an `ostream` class that uses standard output, such as displaying your result or message on the terminal.
3. **err** is an `ostream` class that uses the standard error.
4. **clog** is an `ostream` class that uses the standard unbuffered output.

`cin` and `cout` are commonly used at this stage, so be careful to not confuse them with other operators.

**Code: Use of I/O streams**

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

```
int main() {
   char ch= 'A';
   int a =9,b=10,c;
   cout <<  "Value as output is :  " << ch<< a+b<<endl;
   cout<< "Enter your input integer "<<endl;
   cin>>c;
    cout<< "You entered  "<<c<<endl;
}
```

**Output:**

```
Value as output is: A19
Enter your input integer
3
You entered 3
```

**Code: Use of I/O streams**

```
#include <iostream>

using namespace std;

int main() {
   char ch= 'A';
   int c;
   cout<< "\nEnter your input integer "<<endl;
   cin>>c;
    cout<< "\nYou entered  "<<c<<endl;
    cerr <<  "Error message :  " << ch << endl;
    clog <<  "Error message :  " << c << endl;
}
```

**Output:**

```
Enter your input integer
45

You entered 45
Error message : A
Error message : 45
```

## 2.3 Comments

Commenting in your program will help you keep track of how your program works; program comments serve as notes for future reference, as well.

These are only visible to the program and result in no output change on the terminal.

You can include a single line or multiline comment:

- The **/*** (a slash followed by an asterisk) are special characters; after you type **/*,** you write your comments (consisting of any combination of characters), and follow them with the ***/** characters. This results in a *multi-line comment*.
- For a *single-line comment*, you do not need the ending characters: the **//** (two slashes) are followed by your comment (consisting of any combination of characters).

**Code: Writing Single-Line and Multi-Line Comments**

```
#include <iostream>
 /* this is a multiline comment
 add any notes you need here
 vamios!*/
using namespace std;
 //single line comment made like this
int main() {

   cout<< "\nAdding comments "<<endl;


}
```

**Output:**

```
Adding comments
```

## 2.4 Operators and Types

To perform mathematical operations such as multiplication and addition, we need numbers. We also need operators and operands. Consider the equation 6+7 = 13: the + is the operator, and 6 and 7 (integer values) are our operands.

### 2.4.1 Types of Operators in C++

- *Unary operators* take only one operand, like -8, which signifies the value of negative 8. The ++ operator is an increment operator acting and modifying one value at a time.

- *Binary operators* take in two operands, just as in the mathematical equation where we needed sum of 6 and 7. The insertion (<<) and extraction (>>) operators are also binary operators.
- *Ternary operators* work on three operands.

C/C++ has many built-in operators and can be classified into 6 types:

1. Arithmetic operators are simple and well-known (+, -, *, /, %,++,–). Here, % does not mean "percentage," but "modulus."
2. Relational operators are used for the comparison of the values of two operands (i.e., to check equivalence using greater than or less than, such as ==, >= , and <= )
3. Logical operators are used check for conditions/constraints or complements, giving a result as either true or false in binary 1 or 0.
4. Bitwise operators are used to perform bit-level operations on operands. Any operands you have must first be converted to bit form before applying logic such as AND or OR.
5. Assignment operators are used to assign value to a variable.
6. Other operators can be used, such as `sizeof`, which finds the size of datatypes.

Remember the BODMAS rule: Brackets, Orders, Division, Multiplication, Addition, and Subtraction. The compiler follows the order shown in Table 2.2. Two types of precedence occur in C++ when you encounter an operator i.e. Precedence and Associativity. Here precedence of operator is increasing from bottom to top of the table 2.2 and associativity is referred when two operators of same precedence occur in an expression.

***Table 2.2*** Types Of Operators with Precedence and Associativity

| **Type** | **Operator** | **Associativity** |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |

| Type | Operator | Associativity |
|---|---|---|
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

### Code: Use of Arithmetic Operators

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

int main() {

    int a, b;
    a = 78;
    b = 25;
    cout <<  "a + b =  " << (a + b) << endl;
    cout <<  "a - b =  " << (a - b) << endl;
    cout <<  "a * b =  " << (a * b) << endl;
    cout <<  "a / b =  " << (a / b) << endl;
    cout <<  "a % b =  " << (a % b) << endl;

    return 0;


}
```

**Output:**

```
a + b = 103
a - b = 53
a * b = 1950
a / b = 3
a % b = 3
```

## 2.5 Type Conversion

The process of converting a value from one type to another type is called *type conversion*; there are two kinds of type conversion, implicit and explicit.

### Implicit Conversions

- Done by the compiler on its own using intelligence, without any help from the user.
- Like if a datatype takes a new value different than the present one, so it upgrades it to the value to adjust without giving any errors.

#### Code: Implicit Conversion

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

int main() {

    int a = 34; // integer type
    char b = 'a'; // character type
    a = a + b;//seems illegal butt
    float c = a + 1.0;
    cout <<  "a =  " << a << endl
         <<  "b =  " << b << endl
         <<  "c =  " << c << endl;

    return 0;
}
```

#### Output:

```
a = 131
b = a
c = 132
```

### Explicit Conversions

- It allows us to explicitly tell the compiler to convert a value from one type to another type, and any fault is our responsibilities.
- We will use the static_cast operator- static_cast<new_type>(expression).

#### Code: Explicit Conversion

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

int main()
{
    double d = 1.6;
    int sum = (int)d + 5;
    cout <<  "Sum =  " << sum;
    return 0;
}
```

**Output:**

```
Sum = 6%
```

## 2.6 Keywords

*Keywords* are reserved words with a special meaning associated with them. Therefore, they cannot be used for other purposes in C++ programs. C++ reserves a set of 92 words.

Here is a list of all the C++keywords.

| | | |
|---|---|---|
| alignas | bitand | export reinterpret_cast |
| alignof | bitor | requires (since C++20) |
| and | bool | return |
| and_eq | break | short |
| asm | case | signed |
| auto | catch | sizeof |
| bitand | char | static |
| bitor | char8_t (since C++20) | static_assert |
| bool | char16_t | static_cast |
| break | char32_t | struct |
| case | class | switch |
| catch | compl | template |
| float | concept (since C++20) | this |
| for | const | thread_local |
| friend | consteval (since | throw |
| goto | C++20) | true |
| if | constexpr | try |
| inline | constinit (since | typedef |
| int | C++20) | typeid |
| long | const_cast | typename |
| mutable | continue | union |
| namespace | co_await (since C++20) | unsigned |
| new | co_return (since | using |
| extern | C++20) | virtual |
| false | co_yield (since C++20) | void |
| do | decltype | volatile |
| not | default | wchar_t |
| not_eq | delete | while |
| nullptr | double | xor |
| operator | dynamic_cast | xor_eq |
| or | else | public |
| or_eq | enum | register |
| private | explicit | protected |

## 2.7 Loops in C++

Not every program you write will be unique. You may have a block of code that has to repeated until a specified condition is satisfied, and that is what loops are for in C++. Let us consider an example of printing dates of a particular month (June, in this case). June has 30 days, with dates from 1-30, so we write the program as follows:

```
#Ie <iostream>

int main()
{
    cout << "June!";
    cout << "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
              20…….Pls No";
    return 0;
}
```

Let us examine some of the loops available for this program.

### For Loop

We can print the dates in our date program using a for loop. The general syntax for a for loop is as follows:

```
for (initialization; condition; end condition)
{
   //statement or for body
}
```

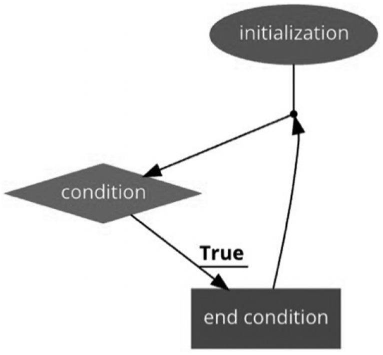


*Figure 2.2* Flow diagram (for loop)

Let's consider the steps in the loop:

1. The initialization is your starting position; in our date example here, it is the first date of the month.

2. The second part is where the condition is checked to able to enter the for loop.
3. Either the terms satisfying the other statements execute or the program comes out of the loop (i.e., it terminates).
4. Lastly, the end condition is used to change the initialization for the next round of loops, often called the next iteration. This usually involves an increment/decrement operation.

Try the following code to print June dates:

```
#include <iostream>

int main()
{
    for (int count= 1; count <= 30; ++count)
        std::cout << count << ' ';

    return 0;
}
```

**Output:**

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 %
```

Here is another example of a for loop that helps us see the inner working of for loops:

```
for (;;)
    statement;
```

**Output:**

The result of this code is an infinite loop that needs to be terminated.

### While Loop

A for loop might be too complex for our program. We do have a simpler loop that involves having only one condition inside of it. The syntax for a while statement is as follows:

```
while (condition)
{
    statement;}
```

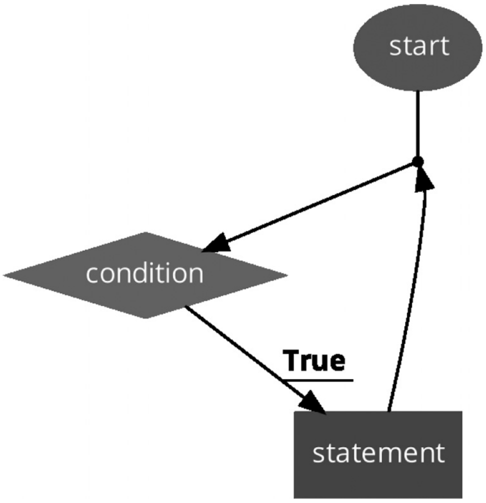


*Figure 2.3* Flow diagram (while loop)

Inside the while loop, as long as the condition is true, we do the work. Otherwise, we exit the loop. In the next example, we use the end condition and start expression inside the while loop's body for the next iteration.

```
#include <iostream>

int main()
{
    int count=1;
    while (count <= 30)
    {
        std::cout << count << ' ';
        ++count;
    }

    std::cout << "done!";

    return 0;
}

while (true)
{
  // this loop will execute forever
}
```

### Do While Loop

The do while loop is similar to the while loop, but here, the work you need to repeat will execute at least once. The syntax for a do while loop is as follows:

```
do
  {statement; // can be a single statement or a compound statement}
while (condition);
```

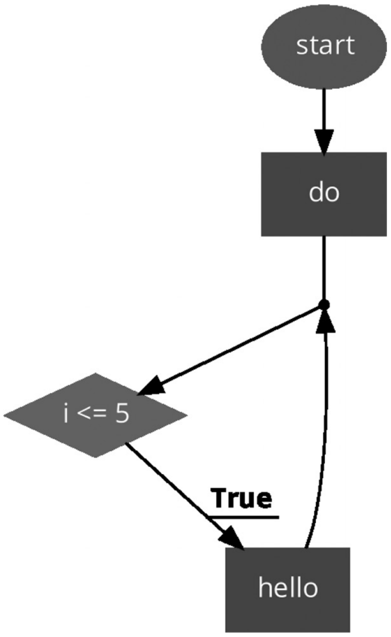


*Figure 2.4* Flow diagram (do while loop)

### Code: Do While Loop Example

```
#include <iostream>

int main()
{
     int i = 1;
          do{
              std::cout<<i<<"\n";
              i++;
          } while (i <= 5) ;
          return 0;
}
```

**Output:**

```
1
2
3
4
5
```

### Nested Loops

A nested loop is a loop within a loop, and this structure allows us to perform many functions more efficiently.

```
#include <iostream>
using namespace std;
int main() {

    for (int i = 1; i <=3; ++i) {

        for (int j = 1; j <=i; ++j)
        {
            cout << "*";
        }
    cout << "\n" << endl;
    }

    return 0;
}
```

**Output:**

```
*
**
***
```

## 2.8 Control Statements

Controlling where your program leads is a quality of a good programmer. In C++, there are statements and types that change the normal flow of code.

A *conditional statement* is a statement that specifies whether some associated statement(s) should be executed or not.

```
if (condition)
    statement;
elseif (condition)
    statement;
```

```
else
    statement;
```

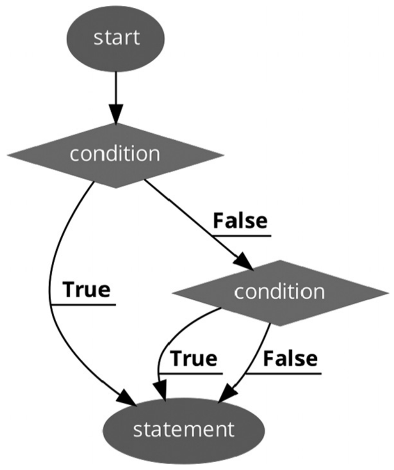


***Figure 2.5*** Flow diagram (if-else)

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;

    if (x > 10)
        std::cout << x << " is greater than 10\n";
    else
        std::cout << x << " is not greater than 10\n";

    return 0;
}
```

*Jump statements* can be used to either pass by a block of code or terminate before the last actual statement of the program. Some of the jump statements and their uses are as follows:

- `continue`*:* This statement will skip the rest of the loop body and continue to the iteration. It will test the condition again before iterating. This statement can be used inside for loop or while or do-while loop.

**Code: Continue Statement**

```
#include <iostream>
```

```
using namespace std;

int main() {
  for (int i = 0; i < 6; i++) {
    if (i == 4) {
      continue;
    }
    cout << i << "\n";
  }
  return 0;
}
```

**Output:**

```
0
1
2
3
5
```

- `break`: Breaking out of a difficult and unwanted situation is important for the code's "health," so this statement terminates the loop and executes the next statement.

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

int main() {
  for (int i = 0; i < 6; i++) {
    if (i == 4) {
      break;
    }
    cout << i << "\n";
  }
  return 0;
}
```

**Output:**

```
0
1
2
3
```

- `return`: You might have noticed this code in the snippets presented here. There is a `return` statement and `int`, so the code returns an integer value.
- `goto`: What if your needed body of code is far away from the ongoing loop? In this situation, use goto to jump directly that block of code. Use this statement with caution.as you might loose flow control of the code and it would be difficult to maintain the code.

**Syntax using goto:**

```
goto label_name;
.
.
.
label_name:
```

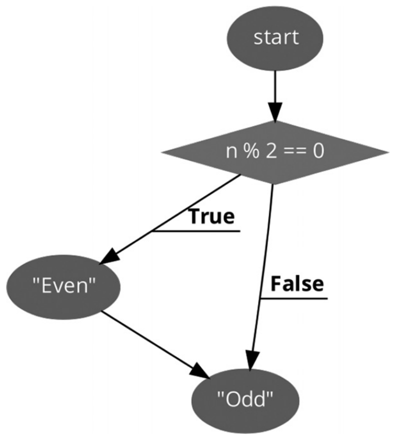


***Figure 2.6*** Flow diagram (goto)

```
#include <iostream>
using namespace std;
int main()
{
    int n = 88;

    if (n % 2 == 0)
        goto place1;
    else
        goto place2;

place1:
    cout << "Even" << endl;
    return 0;//return statement used
place2:
    cout << "Odd" << endl;
}
```

**Output:**

```
Even
```

## 2.9 Defining Functions

In C++, many times, whole code blocks of 30-40 lines are repeated. When this occurs, we can use a *function*, which can accept different inputs and process them with the same code to obtain different results. If you recall our Formula 1 example, the teammates with different sets of skills obtained completely different results; in this instance, the function is like the engine (the common link) that is processed to yield two results.

### 2.9.1 Why Use Functions?

- They reduce code redundancy: one single block of code can be called upon many times, wherever it is needed.
- Easy maintenance, as only one location serves as the function's "home."
- Modularity makes code easy to understand and control.
- Abstraction is supported, as library functions can be used without the extra work of copying the whole function.

C++ has two types of functions: built-in or programmer-defined. In a programmer-defined function, blocks of code for a specific task are given a unique name that is used when the function is being invoked (called into action).

#### Function Declaration

The *function declaration* introduces the function and what it is capable of doing provided the necessary parameters. The compiler gets to know the signature of the function and verifies when it is being called.

The syntax for this is as follows:

```
return_type function_name([ arg1_type arg1_name, ... ]) { code }
```

#### Passing Parameters to Functions

The parameters passed to a function are called actual parameters and the parameters received by a function are called *formal parameters*.

There are two major ways to pass parameters:

- *Pass by Value*: Here, the values of the actual parameters are passed to the function's formal parameters. Different memory locations are provided to both the actual and formal parameters, so any changes made inside the functions are not permanent and are valid inside the function body only.
- *Pass by Reference*: Here, the addresses of the values are passed and so two copies do not exist. The changes remain permanent inside and outside the body of the function. Both the actual and formal parameters are stored in the same memory location.

**Code: Pass by value function**

```
#include <stdio.h>
int max(int a, int b)
{
    if (a > b)
    return a;
    else
    return b;
}
int main(void)
{
    int n = 10, m = 20;
    int mx = max(n, m);

    printf("\nmax is %d\n", mx);
    return 0;
}
```

**Output:**

```
max is 20
```

## 2.10 C vs. C++

*Table 2.3* C & C++ Differences

| C | C++ |
|---|---|
| C was developed by Dennis Ritchie in around 1969 at AT&T Bell Labs. | C++ was developed by Bjarne Stroustrup in 1979. |
| C uses procedural programming. | C++ supports both procedural and OOP paradigms. |

(Contd.)

| C | C++ |
|---|---|
| C is a subset of C++. | C++ is superset of C. |
| C does not support polymorphism, encapsulation, and inheritance. | C++ has support for polymorphism, encapsulation, and inheritance. |
| C does no data hiding and can be easily manipulated. | C++ encapsulation hides the data to ensure that data structures and operators are used as intended. |
| Built-in data types are supported. | Built-in as well as user-defined data types are supported in C++. |
| C is a function-driven language and more attention given to the function. | C++ is an object-driven language. |
| The header file used is `stdio.h`. | `Iostream.h` is used in C++. |
| C does not support function and operator overloading. It also does not have the namespace feature and reference variable functionality. | C++ supports both function and operator overloading; it has the namespace feature and reference variable functionality. |
| Functions in C are not defined inside a structure. | Functions can be defined inside a structure. |
| Reference variables are not supported in C. | Reference variables are supported in C++. |
| The namespace feature is not available. | Namespaces provided by C++. |
| The virtual and friend functions are not supported by C. | The virtual and friend functions are not supported by C++. |
| Direct exception handling is not supported. | Exception handling is supported. |
| Memory management occurs through the `malloc()` and `calloc()` functions | New and delete operators are used for memory allocation and deallocation. |

## Summary

- Literal constants (literals) are unnamed values used directly in code.
- The iostream class can handle both input and output streams.
- cin is an istream class that uses the standard input keys for fetching values. cout is an ostream class that results in the standard output.
- The types of operators in C++ are unary operators (which take only one operand), binary operators (which take in two operands), and ternary operators (which work on three operands).
- The process of converting a value from one type to another type is called type conversion.

- C++ reserves a set of 92 words for its own use called keywords.
- C++ has loops, such as for, while, and do while, to perform repeated tasks.
- The function definition and declaration are done with the signature to call later.
- C++ is an improvement over C, and although there are basic similarities, they have many differences.

## Exercises

### Theory Questions

1. Compare C &C++.
2. Define literals and constants in C++.
3. Discuss various types of operators in C++.
4. How is type conversion possible in C++?
5. Define all the io-based stream classes in C++.
6. What does `int` actually tell in a program?
7. Which is better, a `for` or `while loop`? Discuss your answer using examples.
8. Is a `do while` loop more efficient than a normal `while` loop?
9. Define various control statements with examples.
10. Is it better to use `if-else` or `goto`? Please support your reasoning with examples.
11. Discuss the procedure of using functions in C++.
12. Why do you think modular programming is preferred among programmers?

### MCQ-Based

1. Which of the following features is not provided by C?
   - **a.** Pointers
   - **b.** Structures

**c.** References

**d.** Functions

**2.** Which of the following types is provided by C++ but not C?

**a.** int

**b.** bool

**c.** float

**d.** double

**3.** Which of the following is an entry-controlled loop?

**a.** for

**b.** while

**c.** do-while

**d.** both while and for

**4.** Which of the following operators has left-to-right associativity?

**a.** Unary operator

**b.** Logical not

**c.** Array element access

**d.** addressof

**5.** What is the size of a character literal in C and C++?

**a.** 4 and 1

**b.** 1 and 4

**c.** 1 and 1

**d.** 4 and 4

**6.** What is std in C++?

**a.** std is a standard class in C++

**b.** std is a standard namespace in C++

**c.** std is a standard header file in C++

**d.** std is a standard file reading header in C++

7. What does \a escape code represent?
   - **a.** alert
   - **b.** backslash
   - **c.** tab
   - **d.** form feed
8. What does \t escape code represent?
   - **a.** alert
   - **b.** backslash
   - **c.** tab
   - **d.** form feed
9. Which of the following is not a keyword in C++?
   - **a.** int
   - **b.** break
   - **c.** object
   - **d.** void
10. Which will make a permanent change?
    - **a.** pass by value
    - **b.** pass by reference
    - **c.** pass address
    - **b.** both b & c

**Practical Questions**

1. Write a program in C++ to implement operators.
2. Write a program in C++ to print your hobbies on separate lines.
3. Write a program in C++ to print the sum of three numbers.
4. Write a program in C++ to add, multiply, and subtract two numbers from user-given inputs.
5. Write a program in C++ to calculate the volume of a cylinder with all dimensions entered by the user.

6. Write a program in C++ to find the area and perimeter of a rectangle, square, and circle, with all dimensions entered by the user.
7. Write a program in C++ to swap two numbers using a function.
8. Write a program in C++ to convert temperature from Celsius to Fahrenheit.
9. Write a program in C++ to find the maximum of five numbers entered by the user.
10. Write a program in C++ to find the total and average of four numbers the user enters.

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (c) | **2.** (b) | **3.** (d) | **4.** (c) | **5.** (a) | **6.** (b) | **7.** (a) | **8.** (c) | **9.** (c) | **10.** (d) |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition), (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition), (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++*, (BPB Publications, September, 2020).

### Websites

- Learn CPP, accessed May 2022, *https://www.learncpp.com*
- Geeks For Geeks, accessed May 2022, *https://www.geeksforgeeks.org*
- Word Press, accessed May 2022, *https://wordpress.com*
- Code 2 Flow, accessed May 2022, *https://code2flow.com*
- Silly Codes, accessed May 2022, *https://sillycodes.com*

CHAPTER 3

# ARRAYS AND STRINGS

## 3.1 What is an Array?

An *array* is like a storage facility: it allows you to keep similar data types under one "roof," identified by one name. Items stored inside an array can be integers, characters, and strings. Whatever you wish to group together, you can using an array. Let us see how they are declared and initialized.

### 3.1.1 Ways to Declare Arrays

- `Datatype array_name[array_size];` declared by giving the size
- `Datatype array_name[]={element1, ele2..};` declared by initializing the elements
- `Datatype array_name[array_size]={element1, ele2..};` declared by initializing the elements and size
- `Datatype array_name[array_index]= ele3;` assigning an element to a particular index

Here, the `[]` subscripting operator is used so the compiler knows it is an array; this is followed by similar types of elements, with an index starting from `0` (i.e., the first element of any array is always assigned index 0, `arr[o]`, and the length of an array is n-1, where n denotes the total number of elements). Let's consider an example. Try visualizing an array as a passenger train with three coaches, one behind the other, with ticket numbers assigned from 0 to 2. The array named `Train` has three coaches with the indexes as shown in Figure 3.1.

***Figure 3.1*** The "train" array example

### 3.1.2 Ways to Access Array Members

- Array elements stored in contiguous memory locations are accessed sequentially, too.
- The indexing is important for extracting a particular element.
- The element at index 0 also can be said to be a *pointer* to the array name.
- If you are working with a static/fixed array, then last element is extracted using `index[n-1]`.
- For dynamic arrays, the size is fixed at runtime, so use the `array.length()` method to determine the exact size and indexing.

**Code: Train and Coaches Array Example**

```
#include <iostream>
int main()
{
    int train[3]={10,11}; // declare three coaches
    train= 12;
    std::cout << "\nElements in Train are\n";
    std::cout << "Elements 1: "<<train[0];
    std::cout << "\nElements 2:"<<train;
    std::cout << "\nElements 3: "<<train;
    return 0;
}
```

**Output:**

```
Elements in Train are
Elements 1: 10
Elements 2:11
Elements 3: 12%
```

### 3.1.3 Traversing a 1D Array

Here, *1D refers* to a one-dimensional array which you can easily traverse using the loop statements `for` and `while` to reach and fetch array elements, as well as enable the user to input array elements at the particular index.

**Code: Creating a 1D Array**

```
#include <iostream>
using namespace std;
int main() {

  int num_array[6];
  cout << "Enter six numbers: " << endl;
  for (int i = 0; i < 6; ++i)
  {
    cin >> num_array[i];//take elements from user
  }
  cout << "The numbers are: ";
  for (int j = 0; j < 6; ++j) {
    cout << num_array[j] << "  ";// displaying user array
  }

  return 0;
}
```

**Output:**

```
Enter six numbres:
12
79
45
33
11
23
The numbers are: 12  79  45  33  11  23
```

## 3.2 Operations on an Array

Now that we have learned about making arrays, we can perform various operations on them.

### 3.2.1 Passing an Array to Functions

Functions are very useful, but you can pass array elements that are not needed, which creates a problem: imagine passing thousands of such elements! You can avoid this situation by only passing the name of the array, which acts as a pointer to the first element and whole array. It is important to ensure that the function signature indicates it is an array by using one of the following:

- `int func(int array_name[]){ //body }`
- `int func(int array_name[array_size]){ //body }`
- `int func(int *array_name){ //body }`

**Code: Passing an Array**

```
#include <iostream>

int coach (int train[], int s)
{
int i,price=0;
for( i = 0; i < s; i++)
price += train [i];
return price;
}
int main ( )
{
int price,s;
int train[3]={1050,1100};

std::cout << "How many passengers ";
std::cin >> s;
price = coach (train, s);
std::cout << "\nYour bill is:"<<price;
return 0;
}
```

**Output:**

```
How many passengers 3

Your bill is: 2150 %
HOw many passengers 3

Your bill is:2150%
```

### 3.2.2 Finding the Length

It is important to determine the exact length of a dynamic array as entered by the user, because the size might change midway through the program.

This can be done using the `sizeof` method (which gives us the number of bytes in a particular datatype) by passing the name of the array.

**Code: Using sizeof to Determine Array Length**

```
#include <iostream>
using namespace std;
int main()
{
    int numbers[]={ 0,1,1,2,3,4,5,6,7,7};
    cout << sizeof(numbers)/ sizeof(numbers[0])<< '\n';//gives
             number of elements

    return 0;
}
```

**Output:**

```
10
```

### 3.2.3 Enum in C++

You can make your own datatypes and group them together (like the months). Enumeration in C++ helps the user define data types for versatility. Keep in mind that you can change the indexing midway, as shown in the following code example where "Carlos" (carlos) gets the position value 10, as it succeeds 9.

The syntax for enumeration is as follows:

```
enum enum_Name {elements…,..,};
```

**Code: Enum Example**

```
#include <iostream>
using namespace std;
enum drivers {checo,max,gasly,lewis,yuki=9,carlos};
int main()
{
    drivers d1,d2; //here data type is drivers and variables
                 are d1 and d2
    d1 = carlos ;
    d2= gasly;
    cout << "\nDriver is: " << d1<<endl;//their position in enum
      cout << "\nDriver is: " << d2<<endl;

    return 0;
}
```

**Output:**

```
Driver is: 10
Driver is: 2
```

### 3.2.4 Searching

Accessing each and every element of an array by index (called *traversing*) is not efficient for large arrays, so we should work with an algorithm to search for the element we are looking for. These search methods are as follows.

#### Linear Search

In a linear search, we sequentially search for the element needed by traversing through the array and checking whether the search element matches the array element.

#### Algorithm for a Linear Search

1. Start.
2. Take the size and element input from a user if the array is not given.
3. Input the search element from the user.
4. Initialize the for loop from i=0 to index size-1.
5. Use a nested if loop for the search element == array[i].
6. If the result is true, then a match is found; break out of the loop.
7. If the result is false, then execute the else part of the code.
8. Stop.

#### Code: Example of a Linear Search

```
#include<iostream>
using namespace std;
int main()
{
    int size,e,flag=0;
    cout<<"\nGive the size of the array "<<endl;
    cin>>size;
    int array[size];
    cout<<"Enter the elements of the array "<<endl;
    for(int i=0;i<size;i++)
    {
        cin>>array[i];
    }
    cout<<"What do you wish to search for"<<endl;
    cin>>e;
    for(int i=0;i<size;i++)//traverse array
    {
        if(array[i]==e)//check with element to match
        {
            flag=i;
            break;
        }
    }
    if(flag)
    cout<<"Element "<<e<<" is at index "<<flag;
    else
    cout<<"Element "<<e<<" is not present\n";
    return 0;
}
```

**Output:**

```
Give the size of the array
4
Enter the elements of the array
11
23
66
77
What do you wish to search for
88
Element 88 is not present
```

You can also easily find the index at which the search element was found by making a flag that is `false` at the start and assigning the value 1 to it inside the `if` statement.

### Binary Search

In this search algorithm, we are provided with a sorted array from elements placed in order from the minimum to maximum. Now, the work is divided in half by finding the midpoint about which the next half is found. The program checks whether the search element is greater or smaller than the value being searched for, and this process is repeated until the array becomes empty and we find the matching element.

### Algorithm for a Binary Search

1. Start.
2. Take the size and element input from the user if the array is not given.
3. Input the search element from the user.
4. Call the binary search function and pass the array, size, `firstindex`, and `lastindex`.
5. Execute the nested `if` loop search element == `array[mid]` or < `array[mid]` or> `array[mid]`.
6. If the result is `true`, then the match is found, return mid.
7. If the result is `false`, then return 0 and exit the loop.
8. Stop.

### Code: An Example of a Binary Search

```
#include<iostream>
using namespace std;
int binarySearch(int array[], int a, int b, int s) {
   if (a <= b) {
      int mid = (a + b)/2;
      if (array[mid] ==s)
         return mid ;
      if (array[mid] > s)
         return binarySearch(array,a, mid-1, s);
      if (array[mid] < s)
         return binarySearch(array, mid+1, b, s);
   }
   return 0;
}
int main(void) {
   int array[] = {23,67,45,11,33,44,55};
   int size = sizeof(array)/ sizeof(array[0]);
   int s;
   cout << "Enter the element to search: \n";
   cin >> s;
   int flag = binarySearch (array, 0, size-1, s);
   if(flag){
      cout<< s <<" found at index "<< flag <<" ";
   }else{
       cout<< s <<" is not in the array sorry!";
```

```
    }
    return 0;
}
```

**Output:**

```
Enter the element to search:
55
55 found at index 6 %
```

## 3.3 Multi-Dimensional Array

We learned about 1D arrays and their operations earlier, so let us move on to working with 2D and 3D arrays. These kinds of arrays have rows and columns, and you can express them using the following statements:

2D array: `datatype array_name[rows][column];`

3D array: `int three_d[array_no][rows][columns];`

### Code: An Example Showing 2D and 3D Arrays

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

int main()
{
    int array[3] = {{5,1,4}, {2,3,7}};
    cout<<"\n2D Array:\n";
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << array[i][j]<<" ";
        }
        cout<<"\n";
    }

     int x[3][3] = { { { 7, 1,5 }, { 26, 3,88 }, { 45, 5,78 } },
                     { { 66, 77,33}, { 84, 9,32 }, { 10, 11,12
} } };
 cout<<"\n3D Array:\n";
    for (int m = 0; m < 2; ++m) {
        for (int n = 0; n < 3; ++n) {
            for (int q = 0; q < 3; ++q) {
                cout<< x[m][n][q]<<" ";
            }
```

```
                cout<<"\n";
            }
           cout<<"\n";
        }

        return 0;
    }
```

**Output:**

```
20 Array;
5 1 4
2 3 7

30 Array:
7 1 5
26 3 88
45 5 78

66 77 33
84 9 32
10 11 12
```

## 3.4 Strings

A *string* in C++ is a data type that represents characters that are stored as a collection of bytes in contiguous memory locations. These are useful when we need to work with text. The syntax for strings is as follows:

- `string str_name = "a string";` // this only reads characters until a whitespace is introduced
- `char array[7] ={'c','a','r','l','o','s','\0'}`;//array of characters with the last element always null to identify the end of the string
- `char array[] ={'c','a','r','l','o','s','\0'};`
- `char array[size]="string";`
- `char array[]="string";`

**Code: An Example of Working with Strings**

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

```
int main() {
  string str1;
  cout << "Enter your name: ";
  cin >> str1;
  cout << "Name: " << str1;
  return 0;
}
```

**Output:**

```
Enter your name:
Carlos Sainz
Name: Carlos%
```

## 3.5 String Functions

The C++ string class provides a wide range of functions to perform on strings, as shown in the following table.

*Table 3.1* Syntax and Functionality of String Functions

| Function Syntax | Functionality |
|---|---|
| `strcat(char_arr1, char_arr2);` | Concatenate two character arrays |
| `string1.append(string2);` | Appends a string to another string's end |
| `int length();` | Find the length of a given string |
| `void swap(string& string1);` | Swap two strings |
| `int compare(const string& string1);` | Check whether two strings are equal or not |
| `int size()` | Find the size in bytes of the string. |
| `int find(string& string1, int pos, int size)` | Find a string at a given position |
| `int copy(string& string1)` | Copy contents of one string onto another |
| `int capacity()` | Returns the amount of memory space allocated |
| `getline()` | Read input string until the `\n` character is reached |
| `resize()` | Increase or decrease the string size |
| `end()` | Identify the end of a string |
| `rend()` | Reverse the ending of a string |
| `char& at(int pos)` | Find a character at the position |
| `string& append(const string& string1)` | Append a string to another |

## Summary

- Array is a data type in C++ to store similar data under the same name, and it is accessed using indexes.
- All array elements are stored in contiguous memory locations.
- The element at index 0 can also be said to be a pointer to the array name, and the array goes up to n-1
- An array passes through a function using only its name.
- Enum is a user-defined data type.
- Accessing each and every element by index is called traversing an array.
- The elements of an array can be searched using a linear or binary algorithm.
- An array of an arrays is called a multi-dimensional array.
- A string in C++ is a data type that represents a collection of characters that is stored as a collection of bytes in contiguous memory locations.
- String functions such as concat, compare, length, and size were discussed.

## Exercises

### Theory Questions

1. Discuss the concept of an array in C++.
2. How do you declare an array?
3. What are the advantages and disadvantages of arrays?
4. Explain how 2D and 3D arrays work.
5. Compare the linear and binary search algorithms for arrays.
6. What is meant by dimension and subscript with respect to arrays?
7. What is a string in C++?
8. Compare strings and character arrays.
9. Discuss various operations done on strings in C++ using examples.

**MCQ-Based**

1. Array elements are always stored in _________ memory locations.
   - **a.** Sequential
   - **b.** Random
   - **c.** Sequential and random
   - **d.** None of the above
2. A string is a(n) _________.
   - **a.** data type in C++
   - **b.** array of characters ending with a null character
   - **c.** array of characters starting with a null character
   - **d.** array of integers ending with 0
3. Choose the correct statement about string objects.
   - **a.** They are terminated by a null character "\0."
   - **b.** They have a limited, defined size.
   - **c.** They are dynamic.
   - **d.** None of the above
4. Which of the following is (are) used to find the length of a string?
   - **a.** length()
   - **b.** size()
   - **c.** max_size()
   - **d.** both size() and length()
5. Which of the following is the correct way to declare an array?
   - **a.** int array;
   - **b.** int array[size];
   - **c.** array{ele};
   - **d.** array{ele,ele} array[size];

6. What is the index of the last element of an array having 11 elements?
   a. 12
   b. 10
   c. 0
   d. None
7. In which process do we match every element in an array with a search key?
   a. Linear search
   b. Bubble sort
   c. Merge sort
   d. Binary search
8. What is the length of array {1,1,33,43,22,43,88,99}?
   a. 9
   b. 10
   c. 8
   d. 0

**Practical Questions**

1. Write a program in C++ to find the largest element of a given array of integers.
2. Write a program in C++ to print letters or your name in a string array.
3. Write a program in C++ to find the largest element in an array.
4. Write a program in C++ to find the second largest element in a given array of integers.
5. Write a program in C++ to find the sum of elements in a given array of integers.
6. Write a program in C++ to reverse a string.
7. Write a program in C++ to find the largest word in a string.
8. Write a program in C++ to sort a string alphabetically.

9. Write a program in C++ to count all the characters in a string except spaces and vowels.
10. Write a program in C++ to check if a given string is a palindrome (a word read the same forwards and backwards, like "racecar" or "kayak").
11. Write a program in C++ to find the sum of two matrices.
12. Write a program in C++ to find the transpose of a given matrix.
13. Write a program in C++ to multiply two matrices.

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (a) | **2.** (b) | **3.** (a) | **4.** (d) | **5.** (b) | **6.** (b) | **7.** (a) | **8.** (c) | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition), (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition), (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++*, (BPB Publications, September, 2020).

### Websites

- Learn CPP, accessed June 2022, *https://www.learncpp.com*
- Geeks For Geeks, accessed June 2022, *https://www.geeksforgeeks.org*
- Silly Codes, accessed June 2022, *https://sillycodes.com*

CHAPTER 4

# POINTERS IN C++

## 4.1 Introduction

After you work through this chapter, it is hoped that you might feel a little more confident about using pointers in C++. What exactly is a pointer? In programming terms, a *pointer* is a variable pointing towards the memory space or the address of a given variable. A pointer holds the address of the very first byte of the memory location where it is pointing to. This first byte address is called the *base address*, and it is like a variable used to store the address of another variable.

The concept of pointers takes time and effort to understand, but keep practicing your coding skills while using them. With some effort, pointers can play an important role in helping you make part of your code faster and easier to access. You can do some important actions with pointers, even declaring a pointer to a pointer. Let's start with the basic declaration and initializing process for pointers.

## 4.2 Pointers: Declaration and Initialization

Let's start with the basics. The `&` (ampersand) is used to make reference to the memory location of another variable. Pointers can be *dereferenced*, which we do with the * (asterisk), and this means the operator gets the value of the variable. The following code snippet declares a variable of type pointer.

**Code: An Example of Pointers**

```
#include <iostream>
int main()
{
    int *variable;
    int var=10;
    std::cout << var << '\n';
    std::cout << &var << '\n';
    std::cout << *(&var) << '\n';

    return 0;
}
```

**Output:**

```
10
0×30cc965f8
10
```

In the prior code snippet, the keyword is indicating a valid data type in C++, and `variable` is the name of the pointer variable. When the data type is taken as an integer (`int`), it will only point towards an integer value, i.e., it will be a pointer to an integer. However, a pointer can point to any part of the memory, as this action is its sole purpose. Arithmetic operations to be done are so done with reference to the base type. Therefore, the correct and relevant declaration of pointers in C++ is a priority.

The pointer in the code snippet in this example is that the `var` value is equal to the pointer of the address, and the `&` gives us this number, which is the exact memory location where the variable is stored. Now, let's try visualizing how a pointer works using an image. In Figure 4.1, the cylinders serve as the memory space with addresses "xxx" and "yyy." The arrow shown in the figure acts as the pointer pointing towards the memory location.

There are few more ways to declare a pointer to assign the address of the variable. Keep in mind that changes made to references are permanent and often irreversible. Therefore, we should use pointers with care.

In C++, we deal with *null pointers*. As mentioned in previous chapters, null is not equal to zero, but indicates that the value does not exist. A null or zero pointer points towards an address that is not legal and null. Why use it if no reference to an address can be made? As you proceed on in your coding journey and use pointers more often, you might face an issue where

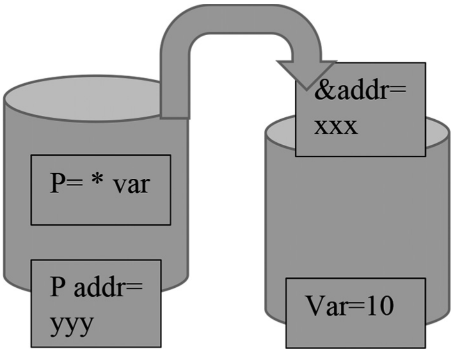


***Figure 4.1*** How a pointer works

you need to check whether the pointer is being used (for example, whether `id=s` is valid). You can do this checking with the help of a null pointer. All you need is to match the variable against a null pointer: if the value holds true, the program will be altered because the pointer used is not legal, helping you to make the relevant changes.

You can declare a null or zero pointer in the two ways: by either assigning a NULL value to a pointer variable or using the keyword `nullptr`, as shown in following example.

### Code: Using the Null Pointer

```
#include <iostream>
int main()
{
    int *ptr1 = nullptr ;//or use = NULL
if(ptr1 != nullptr)
{
   printf("legal address pointed\n");
}
else{
    printf("legal address not pointed\n");
}
    return 0;
}
```

**Output:**

```
legal address not pointed
```

Look at the following example and try to predict the output on your own (or you can test the code on your own computer to see what the output looks like).

**Code: Pointers at Work**

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

int main ()
{
  int val1, val2;
  int * ptr1, *ptr2, ptr3;

 ptr1 = &val1;
  *ptr1 = 10;
  ptr1 = &val2;
  *ptr1 = 20;
  ptr3=50;
  cout << "\nValue One = " << val1<< '\n';
  cout << "Value One address = " << ptr1<< '\n';
  cout << "Value two = " << val2 << '\n';
  cout << "Value One address = " << ptr2<< '\n';
  cout << "Value three = " << ptr3 << '\n';
  cout << "Value three address = " << &ptr3 << '\n';
  return 0;
}
```

Did you predict the output? Let us work through the code together:

1. We declared integer type variables `val1`, `val2`, and `ptr2`. `ptr3` is a variable, too, as no dereference symbol is being used with it.
2. Next, two pointer variables, `ptr1` and `ptr2`, are declared and only the `ptr1` pointer is assigned addresses of value one and value two.
3. Then, integer `ptr3` has the given value.
4. All addresses and values will get printed.

**Output:**

```
Value one = 10
Value one address = 0×308cd65e4
Value two = 20
Value one addresss = 0×0
Value three = 50
Value three address = 0×308cd65cc
```

The output shows how the addresses of two separate variables can be assigned to the same pointer. The pointer of the previous value 1 will show zero address, and then point to a new address of value 2.

## 4.3 Casting and Passing Pointers

### 4.3.1 Typecasting

In C++, we can typecast our variables from one data type to another. We can do the same if we wish to change the assigned data type using pointers. Let's see an example of how this is done. We use `reinterpret_cast`, which is a typecasting operator that helps convert the pointer's data type. All it does is change to a new type, and it does not check if the data the pointer is pointing towards is the same or not. Its syntax is as follows (here, one parameter is used as the pointer variable and returns no value):

```
data_type *var_name =
       reinterpret_cast <data_type *>(pointer_variable);
Return Type
```

Here are some important functionalities of reinterpret_cast:

- Special operator. It works only when the pointer has the same data type.
- Helps in casting into any other legal data type
- When `bool` is used, it will automatically be converted into an integer equivalent, i.e., 1 signifies `true` and 0 signifies `false`.
- Mostly used when bits are on the center stage.

Now consider the following example:

**Code: Use of reinterpret_cast**

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

int main()
{
    int* ptr= new int(69);
    char* cr = reinterpret_cast<char*>(ptr);
    cout << *ptr << endl;
    cout << *cr << endl;
    cout << ptr << endl;
    cout << cr << endl;
    return 0;
}
```

**Output:**

```
69
E
0×7fbbed405ac0
E
```

Here, we have typecasted the integer 69 to its corresponding character type value with the help of the special operator `reinterpret_cast`. Now, the `cr` pointer type has value `E`, which is just the ASCII value of 69.

### 4.3.2 Passing

Passing a pointer to a function can be done in two ways: either by simply passing the pointer or by passing a reference, as was done in some previous examples, with simple data taken as the parameter (in this case, the data is the pointer itself).

In the first method (simply passing the pointer), the memory addresses of the variables are passed as parameters. Any changes made within the function's body will reflect on the original pointer. The following example demonstrates this approach.

**Code: Simply Passing the Pointer**

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

void function_A(int *x, int *y) {
   *x= *x+10;
   *y=*y+*x;

}
int main () {
   int a,b;
   int* num1;
   int* num2;
   num1= &a;
   num2=&b;
   cout<<"\nEnter value of Num1 and Num2:";
    cin>>*num1>>*num2;
   cout << "\nBefore Function Call Num1= " <<*num1 <<" Num2= "
        <<*num2<<endl;
   function_A( num1,num2 );
   cout << "\nAfter Function Call Num1= " <<*num1 <<" Num2= "
        <<*num2<<endl;
   return 0;
}
```

**Output:**

```
Enter value of NUm1 and Num2:4
5
Before Function Call Num1= 4 Num2= 5
After Function Call Num1= 14 Num2= 19
```

In the second approach, passing the pointer by using a reference, the reference variables that are created are passed as arguments. This means that any changes made within the function's body will reflect on the original values, hence, the changes are permanent. The following example demonstrates this approach.

### Code: Passing the Pointer Using a Reference

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

void function_A(int &x, int &y) {
   x= x+10;
   y=y+x;

}
int main () {

   int num1;
   int num2;
   cout<<"\nEnter value of Num1 and Num2:";
    cin>>num1>>num2;
   cout << "\nBefore Function Call Num1= " <<num1 <<" Num2= "
        <<num2<<endl;
   function_A( num1,num2 );
   cout << "\nAfter Function Call Num1= " <<num1 <<" Num2= "
        <<num2<<endl;

   return 0;
}
```

**Output:**

```
Enter value of Num1 and Num2:4
```

```
5
Before Function Call Num1= 4 Num2= 5
After Function Call Num1= 14 Num2= 19
```

We can easily swap numbers using pointers by passing the pointer to the numbers and making a temporary pointer to store value. We then swap the provided pointers and print the results before and after swapping.

**Code: Swapping Numbers Using Pointers**

```
#include <iostream>
using namespace std;
int main () {

   int num1=30, num2=90;
   cout << "\nBefore swapping Num1= " <<num1 <<" Num2= "
        <<num2<<endl;
   swap( num1,num2 );
   cout << "\nAfter swapping Num1= " <<num1 <<" Num2= "
        <<num2<<endl;

   return 0;
}

void swap(int *x, int *y) {
   int* temp;
   temp=x;
    x=y;
    y=temp;
}                                                              ]
```

**Output:**

```
Before swapping Num1= 30 Num2= 90
After swapping Num1= 90 Num2= 30
```

## 4.4 Using Pointers with Arrays

We have learned about the role of a pointer in C++ and how it holds the address of a variable/value. However, can it point to a group of variables and help modify them? That is possible: we assign the pointer to the base address of a group of data in an array. This approach is illustrated in the following code (carefully read the explanation given for the one-dimensional array).

**Code: Using a Pointer with an Array (Example 1)**

```
#include <iostream>
int main()
{
    int array[3] = { 101, 102, 103 };
int *ptr = array;//
std::cout <<"\nBase element : \n"<< ptr<<" array[0]= "<<*(ptr);
std::cout <<"\nSecond element: \n"<< ptr+1 <<" array= "<<*(ptr+1);
std::cout <<"\nThird element: \n"<< ptr+1 <<" array= "<<*(ptr+2);

    return 0;
   }
```

**Output:**

```
Base element :
0×30da805ec array[0]= 101
Second element:
0×30da805f0 array[1]= 102
Third element:
0×30da805f0 array[1] = 103%
```

Here, `int *ptr = array;` will assign the base address of the array, which is the first element, to the pointer ptr. The following arithmetic operations of pointers incrementing `ptr` will point towards the next integer in line, that is, the second element of the array:

```
ptr + 1 = = &array;
ptr + 2 = = &array;……….. or
*ptr = = array[0];
*(ptr + 1) = = array;
*(ptr + 2) = = array;……….
```

All you need to take care of is the arithmetic to be able to access all the elements of the 1D arrays. The same can be done to assign a pointer to all elements of an array, that is, to the array with the following simple syntax:

```
data_type (*pointer_var)[Array_size];
```

Up until now, we have been referencing the base 0th element of the array and accessing other elements from it. However, now the whole array falls under the "umbrella" of one pointer. This method will help us when we work with multi-dimensional arrays, too. How they work is explained in the following example.

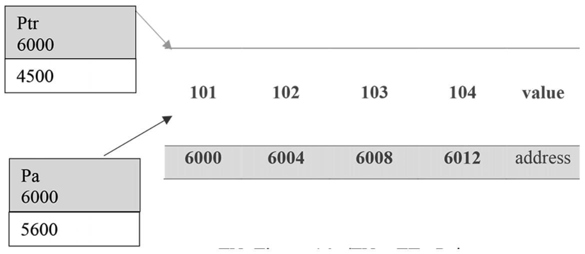


*Figure 4.2* Pointers to an array

In Figure 4.2, `Ptr` is a pointer is pointing to the base element of an array consisting of 4 integers. `Pa` is a pointer pointing to the whole array. Recall the pointer arithmetic, where on incrementing, the `Pa++` pointer will move 16 (4*4) bytes, as denoted by the whole array, while `Ptr++` will result in a step of only 4 bytes. Note that here, dereferencing the `Ptr` pointer will give the whole array represented by the name of the array, which is also the base element.

## Code: Using a Pointer with an Array (Example 2)

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

int main () {

   int array[4];
    int *ptr;
    int *pa;
    pa = array;
    ptr= &array[0];
    cout << "\nptr++ = "<< ptr++ << endl;
    cout << "\npa++ = "<< pa++ << endl;

    cout<<"\nArray addresses with help of pointers: "<< endl;
    for (int i = 0; i < 4; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }
    cout << "ptr++ = "<< ptr++ << endl;
    cout << "pa++ = "<< pa++ << endl;
    return 0;
}
```

**Output:**

```
ptr++ = 0×309abf5a0

pa++ = 0×309abf5a0

Array addresses with help of pointers:
ptr + 0 = 0×309abf5a4
ptr + 1 = 0×309abf5a8
ptr + 2 = 0×309abf5ac
ptr + 3 = 0×309abf5b0
ptr++ = 0×309abf5a4
pa++ = 0×309abf5a4
```

In the case of multi-dimensional arrays, we can also access all the elements using a pointer either to the base element or the whole array. When representing 2D arrays, we use row number `i` and column number `j` as the two parameters for defining a 2D array. We write the 2D array code as `array[i][j]`, so its corresponding pointer representation is `*(*(arr + i) + j)`. Let us consider the following two-dimensional array:

```
int arr[3][4] = { {11, 22, 33, 44}, {52, 65, 75, 88}, {92, 101,
121, 152} };
```

In this example, `*(array+i)` where `i=1`, we obtain the base element of `array[i]`. However, if we take into consideration the 2D array, we increment `+j` to get the address of the `jth` element of the `ith` 1D array. We then have the following:

`*(arr + i) + 1= =` the address of the first element of the `ith` 1D array

`*(arr+i)+2 = =` the address of 2nd element of the `ith` 1D array.

`*(arr + i) + j` will represent the address of the `jth` element of the `ith` 1D array.

**Code: Using a Pointer with an Array (Example 3)**

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

int main ()
{

   int array[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
   int i,j;
   for(i=0;i<3;i++)
   {
```

```
 for(j=0;j<4;j++){
    cout << "*(array+"<<i<<")+"<<j <<"= "<< *(array+i)+j <<endl;

 }}
  return 0;
  }
```

**Output:**

```
*(array+0)+0= 0×3053d95a0
*(array+0)+1= 0×3053d95a4
*(array+0)+2= 0×3053d95a8
*(array+0)+3= 0×3053d95ac
*(array+1)+0= 0×3053d95b0
*(array+1)+1= 0×3053d95b4
*(array+1)+2= 0×3053d95b8
*(array+1)+3= 0×3053d95bc
*(array+2)+0= 0×3053d95c0
*(array+2)+1= 0×3053d95c4
*(array+2)+2= 0×3053d95c8
*(array+2)+3= 0×3053d95cc
```

Similarly, we can create a three-dimensional array where we can access all the elements using three parameters.

## 4.5 Pointer Use

Pointers in C++ have a wide range of use and help perform many tasks with ease. Some of the uses of pointers are listed here.

- Pointers can be passed as arguments to a function by referencing it.
- Array items can be accessed using pointers.
- Pointers help in performing easy arithmetic and swapping operations to values.
- Pointers can be used to return more than one value from a function with ease.
- Pointers help in dynamically allocating memory.
- In C++, pointers help to use data structures efficiently.
- Pointers do valid address checking on values with the help of `nullptr` or zero or a NULL pointer.

*Table 4.1* Pros and cons of pointers

| Advantages | Disadvantages |
|---|---|
| Pointers help in saving memory space. | Even the slightest error in input results in the wrong output |
| Pointers can perform faster executions, as they can directly access and change data from memory. | They work more slowly than other variables. |
| Pointers can be used in file handling. | Always need to be dereferenced |
| Array elements can be accessed using pointers. | If any pointer is made to point to the wrong memory address, it might result in a permanent change to the value there. |
| Pointers help in dynamic memory allocation | There is always the issue of a memory leak if deallocation not handled carefully. |

## Summary

- A pointer holds the address of the very first byte of the memory location where it is pointing.
- This first byte address is called the *base address*, and it is like a variable to store the address of another variable.
- The & (ampersand) is used to make reference to the memory location of another variable.
- A pointer needs to be dereferenced with the * (asterisk) to declare a variable of type pointer.
- `interpret_cast` is a typecasting operator provided to us in C++ that helps convert the pointer's data type.
- A NULL or zero pointer points towards an address that is not legal and null.
- Passing a pointer to a function can be done in two ways: either by passing a value or by passing its reference.
- When passing by value, the copies of pointers are passed as arguments and the changes are reflected in the original variables.
- When passing by reference, the addresses are directly passed as arguments, hence the changes are permanent.
- Pointers also help to access array elements by either the base element reference or the array pointer assigned.

- Pointers in C++ have a wide range of use and help perform many tasks with ease.

## Exercises

### Theory Questions

1. Define a pointer and how it works in C++.
2. How are pointers declared and initialized?
3. What is the usage of the pointer in C++?
4. Discuss pointer arithmetic with a few examples.
5. What is the use of a null pointer in C++? How do we declare it?
6. Write down the difference between an array pointer and base address pointer.
7. How does a null pointer help check address validity in C++?
8. List some drawbacks of using pointers. Support your reasons with examples.
9. What is the difference between a reference and a pointer in C++?

### Practical Questions

1. Write a program to print the address of a variable and input its value from the user.
2. Write a program in C++ to swap two numbers using pointers.
3. Write a program in C++ to swap the first two characters of your name with a pointer as the argument.
4. Write a program in C++ to perform simple arithmetic operations on four different types of pointers.
5. Write a program in C++ for a simple calculator with a menu using only pointer variables.
6. Write a program in C++ to input an array of addresses from a user and print each of these addresses using a pointer.
7. Write a program in C++ to display the difference between passing by value and passing by reference for a pointer variable.

8. Write a program in C++ to find a maximum of five inputs from a user with help from pointers.
9. Write a program in C++ to find an element in an array using pointers.
10. Write a program in C++ to determine the BMI of a person using pointers as arguments for the weight, height, and other factors.
11. Write a program in C++ to display the factorial result of a user-given number, using pointers as arguments.
12. Write a program in C++ to display an array of pointers.
13. Write a program in C++ to print multiples of five as an array of pointers.
14. Write a program in C++ to access elements of a 2D array using pointers.

### MCQ-Based

1. Which operator is used for dereferencing or indirection?

   **a.** *

   **b.** &

   **c.** ->

   **d.** –>>

2. What role does a pointer play in C++?

   **a.** holds the data value

   **b.** holds the name of the variable

   **c.** holds the address of the variable

   **d.** holds the data type of the pointer

3. What is true about the following statement?

   ```
   char* x char y ;
   ```

   **a.** x is a character

   **b.** y is a string

   **c.** x is a character pointer

   **d.** y is a character pointer

**4.** What is true about the following code?

```
int x =1023, y =456;

int *ptr1 = &x, *ptr2 = &y ;

ptr1= ptr2 ;
```

**a.** b = =a

**b.** `ptr1` points to `y`

**c.** `a = =b`

**d.** `ptr2` points to `x`

**5.** Which of the following is the correct way to declare a pointer?

**a.** `int *a`

**b.** `int &a`

**c.** `int ptr`

**d.** All of the above

**6.** What is the method of referencing a value through a pointer?

**a.** Pointer referencing

**b.** Direct Addressing

**c.** Indirection

**d.** Indirect Addressing

**7.** Is `*ptr++` equivalent to `++*ptr`?

**a.** True

**b.** False

**c.** Maybe

**d.** Invalid

**8.** What can pointers helps us with?

**a.** Accessing all array elements

**b.** Dynamic memory allocation

**c.** Implementing data structures

**d.** All of the above

**9.** What does `char****x` mean?

**a.** `x` is a pointer to a pointer to a pointer to a character

**b.** `x` is pointer to a pointer to a pointer to a pointer to a character

**c.** `x` is pointer to a character pointer

**d.** `x` is pointer to a pointer to a character

**10.** Is the `nullptr` the same as an uninitialized pointer?

**a.** Yes

**b.** No

**c.** Maybe

**d.** None of the above

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (a) | **2.** (c) | **3.** (c) | **4.** (b) | **5.** (a) | **6.** (c) | **7.** (b) | **8.** (d) | **9.** (b) | **10.** (d) |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++ (2nd Edition)* (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed July 2022, *https://www.learncpp.com*
- Codes Cracker, accessed July 2022, *https://codescracker.com*
- Geeks For Geeks, accessed July 2022, *https://www.geeksforgeeks.org*
- C Plus Plus, accessed July 2022, *https://cplusplus.com*
- Silly Codes, accessed July 2022, *https://sillycodes.com*

CHAPTER 5

# CLASSES IN C++

## 5.1 Class Making

We talked briefly about what a class is and how in object-oriented programming, we build up from this basic unit. A *class* is a user-defined data type consisting of methods and data grouped together as members (i.e., encapsulated). A class uses objects that store data and have a defined state and behavior; it is also referred to as an *instance of a class*. A class is like a broad category that encompasses some entities that have similar attributes or features. For example, a cat and a tiger have similarities and belong to the *Felidae* animal family. In this example, *Felidae* is the name of the class and cats and tigers are the class members. The following code shows how we declare and use classes in C++.

**Code: Declaring and Using Classes**

```
#include <iostream>
using namespace std;
class Felidae
{
    public:
    string cats;
    string tigers;
    void petName()
    {
    std::cout << "\nCat's pet name is: " << cats;
    std:: cout << "\nTiger's pet name is: " << tigers;
    }
};
int main() {
    Felidae object1;

    object1.cats = "checo";
    object1.tigers = "maxial";
    object1.petName();
    return 0;
}
```

**Output:**

```
Cat's pet name is: checo
Tiger's pet name is: maxial%
```

Let's take a look into how this code works. Try to predict the execution flow to improve your understanding.

1. Although `main` is the typical entry point of execution in a C++ program, here we have declared a class *outside* `main`. It is possible to declare a class either outside or inside `main`.
2. Class `Felidae` is defined with the keyword `class` and members `cats` and `tigers` are strings with the `petName` function.
3. We observe that access specifiers in a class are defined by labels, where public members are accessible to all and private members are only available to the class methods.
4. In `main`, we have taken two objects and accessed the members using the dot operator, and then assigned names to them.
5. One object accesses all data members of the class, so we can create as many objects as we need.

### Class naming tips

- Start with a capital letter and give the class a name that makes sense.
- Assign objects a name that makes it clear what they are for.
- You can use symbols, such as an underscore, in class names.

### Code: Working with Classes

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

```
class Myclass
{
    public:
    int x=40,y;
    void sum()
    {
    std::cout << "\nSum is: " <<x+y;

    }
};
int main() {
    Myclass c1;
    Myclass c2, c3;
    c1.y= 10;
    c2.y = 20;
    c1.sum();
    c2.sum();

    return 0;
}
```

**Output:**

```
Sum is: 50
Sum is: 60%
```

## 5.2 Constructors and Destructors

A class in C++ has data members, like variables of different types and methods (functions). In addition to these, there is another special member function called a *constructor*, whose main role is to initialize a class and construct the value of all data members without accessing and calling all the functions with the object as we did before. Destructors involve "breaking" or eliminating the objects created by the constructor so as to not overload the computer's limited memory.

### Characteristics of Constructors:

1. It has the same name as the class
2. They can be made accessible on a public, private, or protected level.
3. They cannot be inherited or instanced by any other class.
4. The default constructor is called whenever an object is created.
5. In C++, constructors cannot be declared as virtual.
6. They have no return type.

**Code: Using Constructors and Destructors**

```
#include <iostream>
using namespace std;
class Driver
{
    public:
    Driver()
    {
     std::cout << "\nThis is a constructor of class Driver\n"
<<endl;

    }
};
int main() {
    Driver d1;
    return 0;
}
```

**Output:**

```
This is a constructor of class Driver
```

Observe that as soon as the object is created, the constructor of the class is called upon, as it has the ability to be overloaded. Let us now define the various types of constructors.

### Types of Constructors in C++

C++ provides us with three type of constructors, each a little different. These types are as follows.

**1.** Default Constructors

These take up no arguments and all objects of the class are initialized with the same set of values. These are given by the compiler, and they will be 0 or any integer value. If the programmer forgets to define a constructor explicitly in C++, the compiler will provide them with the default constructor implicitly. The syntax for writing the default constructor is as follows:

```
class class_name
 {  class_name()
{ //body
}
};
```

### Code: Using the Default Constructor

```
#include <iostream>
using namespace std;
class People
{
    public:
    People()
    {
    std::cout << "\nThis is a default constructor \n" <<endl;
    }

};
int main() {
    People Object1;


    return 0;
}
```

### Output:

```
This is a default constructor
```

### Code: Implementing Parameterized Constructors

```
#include <iostream>
using namespace std;
class People
{
    public:
    int x;

};
int main() {
    People Object1;
    cout<<"\nWill be provided with any random value:
         "<<object1.x<<endl;
//Garbage value would be shown as x is not initialized

    return 0;
}
```

### Output:

```
Will be provided with any random value: 9015333
```

**2.** Parameterized Constructors

In a *parameterized constructor*, we can pass one or more arguments to the member function so as to assign different initialization values to an object as it is created. Keep in mind that to call this type of constructor, we must use the correct order and type of arguments as defined in the constructor's prototype. The syntax for a parameterized constructor is as follows:

```
class class_name
 {  class_name(argument list..)
      { //body
      }
};
```

**Code: Using Parameterized Constructors**

```
#include <iostream>
using namespace std;
class Student
{
    public:
    Student(string n)
    {
    std::cout << "\nThis is a parameterized  constructor" <<endl;
    std::cout << "\nName is " <<n<<endl;
    }
};
int main() {
    Student s1("Mary");
    return 0;
}
```

**Output:**

```
This is a parameterized constructor
Name is Mary
```

**3.** Copy Constructors

A *copy constructor* helps to create a copy of another object of a class. The copy is created with all the same values for all data members. The syntax for a copy constructor is as follows:

```
class Class_name
 {  Class_name(argument list..)
```

```
        { //body
        }
      Class_name(const Class_name)
        { //body
        }
  };
```

## Code: Using the Copy Constructor

```
#include <iostream>
using namespace std;
class Student
{
    public:
    string n;
    /*Student()
    {
    std::cout << "\nThis is a default constructor \n" <<endl;
    }*/
    Student(string n)
    {
    std::cout << "\nName is " <<n<<endl;
    }
    Student(const Student& s1)
    {
    std::cout << "\nThis is a copy constructor\n" <<endl;

    }
};
int main() {
    //Student s1;
    Student s1("Rio");
    Student s3(s1);

    return 0;
   }
```

**Output:**

```
Name is Rio
This is a copy constuctor
```

## Code: Using Constructors

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

```
{
    public:
    Student()
    {
    std::cout << "\nThis is a default constructor \n" <<endl;
    }
    Student(string n)
    {
   std::cout << "\nThis is a parameterized  constructor" <<endl;
    std::cout << "\nName is " <<n<<endl;
    }
    Student(const Student& s2)
    {
    std::cout << "\nThis is a copy constructor\n" <<endl;
    }
};
int main() {
    Student s1;
    Student s2("Mary ");
    Student s3(s2);
    return 0;
}
```

**Output:**

```
This is a default constructor

This is a parameterized constructor

Name is Mary

This is a copy constuctor
```

Destructors are called when an object is not in use any more and in need of deletion. They help with memory utilization so unused objects don't take up needed space. A destructor is automatically called by the compiler whenever the object seems out of scope. If you wish to do the destruction manually, you use the tilde (~). Let us see some of the destructor's features.

**Characteristics of Destructors:**

1. It has the same name as the class.
2. Helps deallocate the memory of an unused object
3. It takes no parameter and is only of one type, hence, it cannot be overloaded.

**4.** Destruction occurs in the opposite order of construction, meaning the object created last is deleted first (LIFO, Last In First Out).

The syntax for destructors is as follows:

```
class Class_name {  Class_name(argument list..)
      { //Constructor body}
      ~Class_name()
      { //destructor body}};
```

**Code: Using Constructors and Destructors**

```
#include <iostream>
using namespace std;
class Student
{
    public:
    Student()
    {
    std::cout << "\nThis is a default constructor \n" <<endl;
    }
    Student(string n)
    {
   std::cout << "\nThis is a parameterized constructor" <<endl;
    std::cout << "\nName is " <<n<<endl;
    }
    Student(const Student& s2)
    {
    std::cout << "\nThis is a copy constructor\n" <<endl;
    }
    ~Student()
    {
    std::cout << "\nThis is a Destructor for all\n" <<endl;
    }
};
int main() {
    Student s1;
    Student s2("Mari");
    Student s3(s2);

    return 0;
}
```

**Output:**

```
This a default constructor
This is a parameterised constructor
Name is Aditi

This is a copy constructor
This is a Destructor for all
This is a Destructor for all
This is a Destructor for all
```

***Table 5.1*** Constructors & Destructors Difference Table

| CONSTRUCTOR | DESTRUCTOR |
|---|---|
| Helps with allocating memory and initializing values of an object | Helps with deallocating memory of an unused object |
| `Class_name(argument list..)` | `~Class_name()` |
| LIFO (Last In First Out) is followed in calling | LIFO (Last In First Out) is followed in calling |
| Parameterized and copy constructors do take parameters. | It takes no parameter. |
| Can be overloaded | Cannot be overloaded |
| The same class can have more than one constructor. | There can exist only one destructor for a given class. |

## 5.3 The This Pointer

Every object of a class carries with it a copy of all data members, but they access the original copy of the functions. A situation might occur when multiple objects are accessing the same member function. How will the values inside the function be updated? This problem can resolved using the `this` pointer. It stores the address of an object or class instance to enable the member function to update the correct object values. It can be used in various places and in various ways. It can sometimes even be hidden from us, as when the compiler comes across any member function, it will implicitly add `this` as a non-static function parameter to keep track of the address to "recall" which object called this function.

The `this` pointer will always point towards the object or instance being worked on currently. We can explicitly make reference to the `this` pointer by using it inside a constructor or methods and point to the instance variables to be updated without changing the pointer direction.

### Code: Using the this Pointer

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

```
class Average {
    private:
        int num1;
        int num2;
        int r;
    public:
       Average (int num1, int num2) {
            this->num1 = num1;
            this->num2 = num2;

        }
        void AvgResult() {
               cout<<"\nAverage of num1 and  num2 = "<<(this-
>num1+this->num2)/2<<endl;
        }
};

int main () {
    Average object1(22, 33);
    object1.AvgResult();
    return 0;
}
```

**Output:**

```
Average of num1 and num2 = 27
```

## 5.4 Class Methods

We have discussed the class and its members, as well as the special class function constructors. Now, let us consider class methods. Class functions can either be defined inside or outside the class definition. Calling of these functions is carried out using objects of the class through the dot or selection operator.

### Code: Using the Class Method

```
#include <iostream>
using namespace std;
class myclass
{
   public :
   static int num1;
   int y=100,r;
   int sum(int x,int y)
   {
      cout << "\nSum is = " << x+y<<endl;
   }
   static int assign(int x)
   {
      cout << "\nAssigned value to integer= " << x<<endl;
   }
   void avg()
   {
      r= (num1+y)/2;
      cout << "\nAverage is = " << r<<endl;
      cout << "\nName is  " << n<<endl;
   }
   private:
   string n="Joey";

};
int myclass::num1=4560;
int main()
{
   myclass obj1,obj2,obj3;
   myclass::assign(45);
   obj2.avg();
   obj2.sum(20,27);
   cout << "\nThe static variable value : " <<obj1.num1<<endl;
   return 0;
}
```

**Output:**

```
Assigned value to integer= 45
Average is = 2330
Name is Joey
Sum is = 47
The static variable value : 4560
```

In the previous code, you might have observed a new symbol (`::`), which is called a *scope resolution* operator in C++. Its primary role is to access or assign value to static members of a class.

### Uses of the Scope Resolution Operator

- To access global variables to verify if they have name "clashes" with any of the local variables
- To define a particular function body outside the class

- To access static data type members
- Used in inheritance

C++ has a feature that enables inline functions in a class, where the compiler copies the code of a function body whenever it is called upon. This produces faster results, and so we, as programmers, do not have to write code for it again and again. The syntax for calling inline functions is as follows:

```
inline returnType function_Name(Argument List….)
{ // function body}
```

## 5.5 The static Keyword

You might have heard the word static many times. In physics, it mostly defined as a state of rest and no motion, but in C++, the `static` keyword declares any variable, data members, or functions as a type of constant. These values cannot be modified. It is initialized only once and only one copy is used throughout the lifetime of a program.

### Characteristics of the Static Keyword

1. Static variables are initialized only once in a C++ program.
2. A static variable can either be defined within a particular function or outside it.
3. The scope of a static variable is local to the block where it is used.
4. Zero is the default value provided to a static variable if it is not assigned by the programmer.
5. The lifespan of a static variable lasts until the termination of a program, and then the memory space freed.
6. Static functions are called from the class name directly.

The syntax for static functions and variables is as follows:

```
static DataType var = 10; // gives a static variable
static returnType function // static function declared
{ // body
}
```

### Code: Using Static Functions and Variables

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

```
class myclass
{
   public :
   static int num1;
   static int assign(int x)
   {
      cout << "\nAssigned value to integer= " << x<<endl;
   }
};
int myclass::num1=4560;
int main()
{
   myclass obj1;
   myclass::assign(87);
   cout << "\nThe static variable value : " <<obj1.num1<<endl;
   return 0;
}
```

**Output:**

```
Assigned value to integer= 87
The static vaiable value : 4560
```

## 5.6 Memory Management and Garbage Collection in C++

C++ provides other useful operators, like `new` and `delete`. These operators help enhance our flexibility to allocate or deallocate memory whenever required than C functions like `malloc()`, `calloc()`, and `free()`.

Memory management in programming is important, as the RAM space provided in a device is limited. *Garbage collection* is a type of memory management technique done either manually or automatically by the garbage collector. We have done this type of work before: all global and static variables only live until the end of the program, after which they are of no use and the memory is freed up. Local variables inside a function, too, only "live" between the span of the function call and return statement. All of this work is done automatically in C++, and all of it is done at the compile time. C++ also has the ability to allocate memory for variables at runtime, which is referred to as *dynamic memory allocation*. This has to be done manually, unlike as is done in some other programming languages like Java or Python, where the compiler automatically manages the memory allocation task.

### Why Use Dynamic Memory Allocation?

1. Useful for situations where we are not aware of the size of a particular data type until runtime
2. To make a group of data types more flexible and modifiable by the user
3. User input given more attention and results become more personalized

### New Operator

The new operator in C++ helps in dynamically allocating memory . We make a request for memory space and if the required amount of memory is there, then the specified amount of memory is allocated and returns a pointer to it (or null, if it failed to allocate). `Sizeof` can be used to compute its size. The syntax for this operator is as follows:

```
Pointer = new dataType;
Datatype *new datatype [size in int];
pointerVar = new Datatype[int size];
```

### Delete Operator

The `delete` operator in C++ helps in dynamically deallocating memory. The address becomes invalid, and the memory is now used by different data, so it returns void. If an object is allocated using the `new` operator, then it can only be deleted using the `delete` operator. This operator ensures safe and efficient memory use. The syntax for the `delete` operator is as follows:

```
delete ptr_var;
```

### Code: Using the Delete Operator

```
#include <iostream>
using namespace std;
int main () {
   int *ptr1 = nullptr;
   ptr1 = new int;
   *ptr1 = 28;
   cout << "\nValue of pointer variable 1 : " << *ptr1 << endl;
   delete ptr1;
   return 0;
}
```

**Output:**

```
Value of pointer variable 1 : 28
```

## Summary

- A *class* is a user-defined data type consisting of methods and data grouped together as members.
- A *constructor*'s main role is to initialize a class, and it constructs the value of all the data members.
- Three types of constructors are default, parameterized, and copy.
- Destructors help in deallocating the memory of an unused object.
- Constructors and destructors both have the same name as of the class.
- The `this` pointer in C++ that stores the address of an object or class instance enables the member function to update the correct object values.
- Class functions can either be defined inside or outside the class definition.
- Calling class functions is carried out using objects of the class through the dot or selection operator.
- In C++, the `static` keyword declares any variable, data members, or functions as a type of constant. The values cannot be modified.
- `::` is the scope resolution operator in C++, and its primary role is to access or assign value to static members of a class.
- The `new` and `delete` operators are used for memory allocation and deallocation in C++.

## Exercises

### Theory Questions

1. What is a class? Describe the syntax for declaring a class with examples.
2. Discuss class making and declaration in C++?
3. What is the difference between member functions defined inside and outside the body of a class?
4. Explain the different methods of passing object parameters.
5. Discuss constructors and destructors in C++ using examples.

6. How is memory allocation done dynamically in C++?
7. In how many ways can the `static` keyword be used in C++? Discuss this using code examples.

**Practical Questions**

1. Write a program to create a class named `Student` with a string variable name and an integer variable `roll_no`. Assign the value of `roll_no` as 2, and that of the name as `John` by creating an object of the class `Student`.
2. Write a program in C++ illustrating a class declaration and definition, as well as accessing class members.
3. Write a program in C++ to depict the calling of the constructors of a class.
4. Write a program in C++ to depict the calling of the destructors for a class.
5. Write a program in C++ for a simple calculator with a menu using concept of classes.
6. Write a program in C++ to depict the ways of using the `static` keyword.
7. Write a program in C++ to count and display the number of times an object is created using the `static` keyword.
8. Write a program in C++ to depict the use of the new and `delete` operators.
9. Write a program in C++ for a simple calculator with a menu using only pointer variables.

**MCQ-Based**

1. Which of the following statements is correct about classes?

   **a.** An object is an instance of its class.

   **b.** A class is an instance of its object.

   **c.** An object is the instance of the data type of that class.

   **d.** Both A and C.

**2.** If a local class is defined in a function, what is true for an object of that class?

**a.** The object can be accessed, declared, and used locally in that function.

**b.** The object must be declared inside any other function.

**c.** The object is temporarily accessible outside the function.

**d.** The object can call all the other class members anywhere in the program.

**3.** Which of the following refers to the wrapping of data and its functionality into a single individual entity?

**a.** Modularity

**b.** Abstraction

**c.** Encapsulation

**d.** None of the above

**4.** When `struct` is used instead of the keyword class means, what will happen in the program?

**a.** access is public by default

**b.** access is private by default

**c.** access is protected by default

**d.** access is denied

**5.** Which is used to define the member of a class externally?

**a.** :

**b.** ::

**c.** #

**d.** !!$

**6.** What does a class in C++ hold?

**a.** data

**b.** functions

**c.** both data and functions

**d.** arrays

**7.** In a class, data members are also called _________

**a.** Abstracts

**b.** Attributes

**c.** Properties

**d.** Dimensions

**8.** How many access specifiers are present in a class in C++?

**a.** 2

**b.** 1

**c.** 4

**d.** 3

**MCQ**

| Answer Key | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1**. (a) | **2.** (a) | **3.** (c) | **4.** (a) | **5.** (b) | **6.** (c) | **7.** (b) | **8.** (d) | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Master*ing C++ (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed July 2022, *https://www.learncpp.com*
- Codes Cracker, accessed July 2022, *https://codescracker.com*
- Geeks For Geeks, accessed July 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed July 2022, *https://www.udacity.com*
- Scaler, accessed July 2022, *https://www.scaler.com*
- C Plus Plus, accessed July 2022, *https://cplusplus.com*

CHAPTER 6

# INHERITANCE

## 6.1 Introduction

We discussed object-oriented programing concepts briefly in the first chapter. Now, let's learn a few of them in depth. We are familiar with class making and how objects access them to use the members of a particular class. In C++, we can make multiple classes in one program (and make connections between these classes) and enable them to share access to their individual data members. This concept of establishing a relationship between classes where some of the properties are shared is referred to as *inheritance*. This chapter examines inheritance and its type and how you can make full use of this object-oriented programing concept. All topics discussed have code examples, and readers are encouraged to try them all to improve their understanding of this topic.

## 6.2 Inheritance

Just as a child inherits not only the assets of their parents but also some unique features, like eye or hair color, inheritance in C++ is defined as the ability of a "child" class to inherit features of its "parent" class. Let's learn a little more about these terms before we proceed.

- *Parent class*: Also known as the *base class* or *super class.* You can think of it as the entity that existed before its children. All of its children inherit properties from it.

- *Child class*: Also known as the *derived class* or *sub class*. This is a class that is made out of the certain features of the base class.

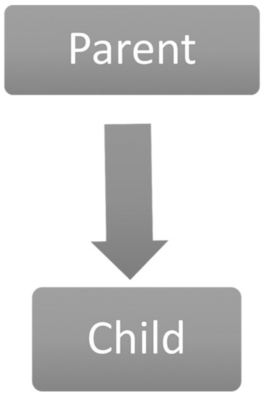


***Figure 6.1*** Inheritance

In Figure 6.1, the arrow going downward shows the relationship between the classes. Can you identify the base class and sub class? The parent is the base class, and the child is a derived class. Let's declare and define the relationship shown in Figure 6.1. The syntax is as follows:

```
class Child : access_specifier Parent
{
// class body
};
```

**Code: Working with Inheritance**

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p=60;
    void showP()
    {
        cout<<"\nParent age is " <<p<< endl;
    }
};
class Child : public Parent
{

   public:
   int c=20;
```

```
    void showC() {
        cout<< "\nChild age is " <<c<< endl;
    }
};

int main() {
    Child c1;
    c1.showP();
    c1.showC();
    return 0;
}
```

**Output:**

```
Parent age is 60
Child age is 20
```

**Code Debriefing**

- As you can observe from the preceding code, we have established a relationship between two classes using inheritance.
- The child class will now be able to inherit and even modify properties from the parent class.
- The objects of the child class can access the member functions of the base or parent class.

### 6.2.1 Access Specifiers

In inheritance, the accessibility of a class has to the data members of another class is defined by the access specifiers. The levels of access given are either directly to the data members or to the inherited class declaration. There are three primary access modes available in C++.

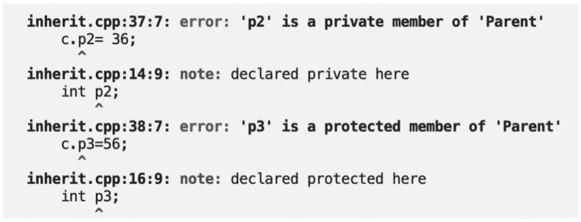
**Direct Data Members**

1. *Public:* Data members declared public can be accessed by any class, irrespective of the relationship its class has with other classes.
2. *Private:* Data members declared private have limited access available to only the base, friend, and derived classes.The concept of Friend classes is to be discussed later.
3. *Protected:* Data members declared protected can be accessed by the base class members and friend classes.

## Code: Using Access Specifiers (Public, Private, and Protected)

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p1=60;
   void showP()
    {
        cout<<"\nParent age is public " <<p1<< endl;
        cout<<"\nParent age is private " <<p2<< endl;
        cout<<"\nParent age is protectd " <<p3<< endl;
    }
    private:
    int p2=12;
    protected:
    int p3=16;
};

class Child : public Parent
{

   public:
   int c1=20;
    void showC() {
        cout<< "\nChild age is public " <<c1<< endl;
        cout<< "\nChild age is private " <<c2<< endl;
        cout<< "\nChild age is protected " <<c3<< endl;
    }
    private:
    int c2;
    protected:
    int c3;
};

int main() {
    Child c;
    c.p2= 36;
    c.p3=56;
    c.showP();
    c.showC();
    return 0;
}
```

**Output:**



```
inherit.cpp:37:7: error: 'p2' is a private member of 'Parent'
    c.p2= 36;
      ^
inherit.cpp:14:9: note: declared private here
    int p2;
        ^
inherit.cpp:38:7: error: 'p3' is a protected member of 'Parent'
    c.p3=56;
      ^
inherit.cpp:16:9: note: declared protected here
    int p3;
        ^
```

Carefully study the given code snippet and the errors that resulted as we tried to access data members declared as private and public in the derived class calls.

### 6.2.2 Inheritance Modes

1. *Public:* If a class is inherited as public, then the access defined for the data members in the base class remain the same in the derived class, too. That is, the public member has public access, the private member has private access, and the protected member has protected access. This is a commonly used method for inheritance access.

   **Code: Giving Access to Data**

```
#include <iostream>
using namespace std;
class parentc
{
    public :
        int agep=52;
    protected :
        int avgSal= 43000;
    private :
        int adharDigit=876;

};
class PublicChild : public parentc
{
  public:
  int pro()
  {
    return avgSal;
    }
};
```

```
int main()
{
    PublicChild obj1;
    cout << "\nPrivate : adharnumber digit is inaccessible as
            it is not inherited "  << endl;
    cout << "\nProtected<->Protected :avg salary is "
         << obj1.pro() << endl;
   cout << "\nPublic<->Public : age is " << obj1.agep << endl;
    cout<< "\n";

    return 0;
}
```

**Output:**

```
Private: adharnumber digit is inaccessible as it is not inherited
Protected<->Protected :avg salary is 43000
Public<->Public : age is 52
```

### Code Debriefing

- As you can see from the code, we have established a relationship between two classes using public inheritance.
- The public child class will now be able to inherit and even modify properties from the parent class members, as their access is declared in the base class.
- An object of the child class can access all the members except the private ones of the base or parent class.

2. *Private:* If a class is inherited as private, then the access defined for the data members in the base class changes to private in the derived class. That is, all public, protected, and private members have private(inaccessible) ranks.

### Code: Using Private Classes

```
#include <iostream>
using namespace std;
class parentc
{
    public :
        int agep=52;
    protected :
        int avgSal= 43000;
    private :
        int adharDigit=876;

};
class PublicChild : public parentc
{
  public:
  int pro()
  {
    return avgSal;
    }
};

int main()
{
    PublicChild obj1;
    cout << "\nPrivate : adharnumber digit is inaccessible as
               not inherited "  << endl;
    cout << "\nProtected<->Protected :avg salary is "
         << obj1.pro() << endl;
    cout << "\nPublic<->Public : age is " << obj1.agep << endl;
    cout << "\n";

    return 0;
}
```

**Output:**

```
Private: adharnumber digit is inaccessible as not inherited
Protected<->Protected :avg salary is 43000
Public<->Public : age is 52
```

### Code Debriefing

- In this code, we have established a relationship between two classes using private inheritance.
- The private child class will now be able to inherit and even modify properties from the parent class members as their access is declared in the base class.
- The objects of the child class can access all the members except the private ones of the base or parent class.
- The new access specifier assigned to all of the inherited material is now private.

3. *Protected:* If a class is inherited as protected, then the access defined for the data members in the base class changes to protected in the derived class, too. That is, public, protected members have protected access ranks and private access remains private (inaccessible).

**Code: Using Protected Inheritance**

```
#include <iostream>
using namespace std;
class parentc
{
    public :
        int agep=52;
    protected :
        int avgSal= 43000;
    private :
        int adharDigit=876;

};
class ProtectChild : protected parentc
{
  public :
  int pro()
  {
    return avgSal;
    }
   int pub()
   {
       return agep;
   }

};

int main()
{
    ProtectChild obj1;
    cout << "\nPrivate : adharnumber digit is inaccessible "
         << endl;
    cout << "\nProtected<->Protected : avg salary is "
         << obj1.pro() << endl;
    cout << "\nPublic<->Protected: age is " << obj1.pub()
         << endl;
    cout<<"\n";


    return 0;
}
```

**Output:**

```
Private : adharnumber digit is inaccessible
Protected<->Protected : avg salary is 43000
Public<->Protected: age is 52
```

### Code Debriefing

- In this code, we have established a relationship between two classes using protected inheritance.
- The protected child class will now be able to inherit and even modify properties from all parent class members as their access is declared in the base class.
- Objects of the child class can access all the members except the private ones of the base or parent class.
- The new access specifier now assigned to all of the inherited material is protected.

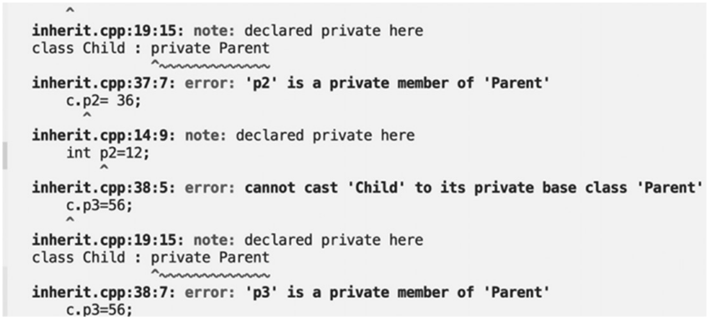
The following code blocks are a few more examples to explain the types of declared inheritance in C++. Try these out on your own to get a better understanding of the concepts we discussed.

### Code: Using Different Types of Inheritance

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p1=60;
   void showP()
    {
        cout<<"\nParent age is public " <<p1<< endl;
        cout<<"\nParent age is private " <<p2<< endl;
        cout<<"\nParent age is protecetd " <<p3<< endl;
    }
    private:
    int p2=12;
    protected:
    int p3=16;
};
```

```
class Child : private Parent
{

   public:
   int c1=20;
    void showC() {
        cout<< "\nChild age is public " <<c1<< endl;
        cout<< "\nChild age is private " <<c2<< endl;
        cout<< "\nChild age is protected " <<c3<< endl;
    }
    private:
    int c2;
    protected:
    int c3;
};

int main() {
    Child c;
    c.p2= 36;
    c.p3=56;
    c.showP();
    c.showC();
    return 0;
}
```

**Output:**



```
        ^
inherit.cpp:19:15: note: declared private here
class Child : private Parent
              ^~~~~~~~~~~~~~
inherit.cpp:37:7: error: 'p2' is a private member of 'Parent'
    c.p2= 36;
      ^
inherit.cpp:14:9: note: declared private here
    int p2=12;
        ^
inherit.cpp:38:5: error: cannot cast 'Child' to its private base class 'Parent'
    c.p3=56;
    ^
inherit.cpp:19:15: note: declared private here
class Child : private Parent
              ^~~~~~~~~~~~~~
inherit.cpp:38:7: error: 'p3' is a private member of 'Parent'
    c.p3=56;
```

Carefully study the given code snippet and errors that resulted when we tried to access the inheritance declared as private in the derived class calls. The access rules might seem difficult to understand, as their combination creates a nine-member set of how class and class members can be inherited.

*Table 6.1* Mapping of inheritance types

| **Inherited as Public** | | | |
|---|---|---|---|
| **Base Class** | | **Derived Class** | |
| Public | | Public | |
| Private | | Inaccessible | |
| Protected | | Protected | |
| **Inherited as Private** | | **Inherited as Protected** | |
| Base Class | Derived Class | Base Class | Derived Class |
| Public | Private | Public | Protected |
| Private | Inaccessible | Private | Inaccessible |
| Protected | Private | Protected | Protected |

## 6.3 Types of Inheritance

Inheritance in C++ can be done in various forms and combinations. There are six types of inheritance that can be implemented to define relationships between various classes.

**1.** Single Inheritance

*Single inheritance* is the most basic type of inheritance, where there is only one parent and one child inherits the properties. You can think of it as a stepladder with two steps connected to one another. The syntax for single inheritance is as follows:

```
class parent
{
    // class body
};
class child : accessMode  parent
{
    // class body
};
```

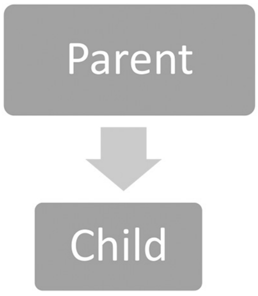


*Figure 6.2* Single inheritance

### Code: Working with Single Inheritance

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p=60;
    void showP()
    {
        cout<<"\nParent age is " <<p<< endl;
    }
};
class Child : public Parent
{

   public:
   int c=20;
    void showC() {
        cout<< "\nChild age is " <<c<< endl;
    }
};

int main() {
    Child c1;
    c1.showP();
    c1.showC();
    return 0;
}
```

### Output:

```
Parent age is 60
Child age is 20
```

### Code Debriefing

- As you can observe from the preceding code, we have established a relationship between the two classes using single inheritance.
- Here, only the child class will now be able to inherit and modify properties from all members of the parent class, as their access is declared in the base class.
- Objects from the child class can access all the members except the private ones of the base or parent class.

**2.** Multilevel Inheritance

*Multilevel inheritance*, as the name suggests, has multiple levels of property sharing. The bottom level does not directly inherit the properties, but still can access all of the top levels' data members. You can think of it in terms of family relationships. This type of inheritance is similar to how a grandchild often has a few features from his grandfather. Objects created only for the grandchild will be able to access all of its preceding classes. The syntax for multilevel inheritance is as follows:

```
class A
{
    // class body
};
class B : accessMode  A
{
    // class body
};
class C : accessMode B
{
    // class body
   };
```

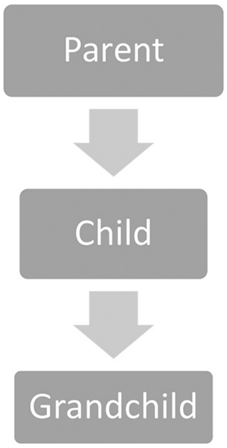


***Figure 6.3*** Multilevel inheritance

**Code: Using Multilevel Inheritance**

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p=60;
   void showP()
    {
        cout<<"\nParent age is  " <<p<< endl;

    }

};

class Child : public Parent
{

   public:
   int c=20;
    void showC() {
        cout<< "\nChild age is " <<c<< endl;

    }

};
class Grandchild : public Child
{

   public:
   int g=4;
    void showg() {
        cout<< "\nGrandchild age is  " <<g<< endl;
        ;
    }

};

int main() {
    Grandchild g1;
    g1.showP();
    g1.showC();
    g1.showg();
    return 0;
}
```

**Output:**

```
Parent age is 60
Child age is 20
Grandchild age is 4
```

### Code Debriefing

- As you can see from the code, we have established a relationship between three classes using multilevel inheritance.
- Here, the grandchild class will now be able to inherit and modify properties from the parent and child class, as their access is declared in the base class.
- Objects from the grandchild class can access all the members except the private ones of the base or parent class.

**3.** Multiple Inheritance

*Multiple inheritance* is when a class inherits properties from two other classes. It has many uses, and the objects created for the bottom-level class can access all members of both of the top classes. The syntax for multiple inheritance is as follows:

```
class A
{
    // class body
};
class B
{
    // class body
};
class C : accessMode A,  accessMode B
{
    // class body
};
```

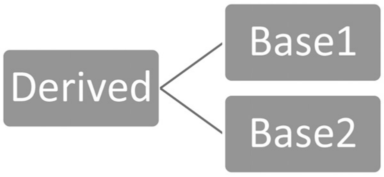


***Figure 6.4*** Multiple inheritance

### Code: Working with Multiple Inheritance

```
#include <iostream>
using namespace std;
class ring1
{
    private :
        string ss;
        float cost;
    public:
    void getdata()
    {
        cout<<"Enter Seat Section & Cost: "<<endl;
        cin>>ss>>cost;
    }
    void showdata()
    {
      cout<<"Seat : "<<ss<<endl;
      cout<<"Cost: "<<cost<<endl;
    }
};
class ring2
{
  private:
    float spectators[3];
  public:
  void getdata()
{
    int i;
    for(i=0;i<3;i++)
    {
        cout<<"\nEnter "<<i+1<<" Section Spectators ";
        cin>> spectators[i];
    }
}
void showdata()
{
    int i;
    for(i=0;i<3;i++)
    {
        cout<<"\n"<<i+1<<"Section Spectators= "<<spectators[i];

    }
}

};

class arena:public ring1,public ring2
{
  private :
  int guest;
  public:
  ring1 r1;
  ring2 r2;
  void getdata()
  {
      r1.getdata();
      cout<<"Enter guest";
      cin>>guest;
      r2.getdata();
  }
  void showdata()
  {
      cout<<"\nGuest count= "<<guest<<endl;
      r2.showdata();
  }
};

int main()
{
    arena obj1;
    obj1.getdata();
    obj1.showdata();


  return 0;
}
```

**Output:**

```
Enter Seat Section & Cost:
4
3500
Enter guest4

Enter 1 Section Spectators 230

Enter 2 Section Spectators 345

Enter 3 Section Spectators 445

Guest count= 4
1Section Spectators= 230
2Section Spectators= 345
3Section Spectators= 445%
```

### Code Debriefing

- In this code, we established a relationship between three classes using multiple inheritance.
- Here, the child class will be able to inherit and even modify properties from the two parent classes, as their access is declared in the base class.
- The objects of the child class can access all the members, except the private ones of the base or parent class.

**4.** Virtual Inheritance

*Virtual inheritance* comes into play when we have multiple inheritance, but the two classes inheriting from one super class in different base classes have the same name. Now, the child of these two base classes will have two copies of the super class methods from both the parents, which is referred to as the *diamond problem*. The diamond problem occurs when an object accessing a method by a name will get confused as to which inherited method of the super class is being called. The solution to this problem is using the virtual inheritance feature in C++, where the two parents inherit virtually from the super class.

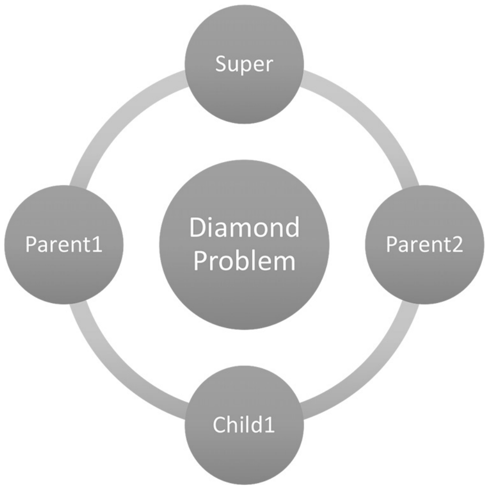


***Figure 6.5*** Virtual inheritance

The syntax for virtual inheritance is as follows:

```
class A
{
    // class body
};
```

```
class B :  accessMode virtual  A
{
    // class body
};
class C : accessMode  virtual A
{
    // class body
};
class D : accessMode B, accessMode C
{
    // class body
};
```

## Code: Working with Virtual Inheritance

```
#include <iostream>
using namespace std;
class pg
{
   public:
   int age;
   void display1()
   {
      cout<<"Enter your age"<<endl;
      cin>>age;
   }

};

class student : public virtual pg
{
    public :
    string name;
    void display2()
    {
        cout<<"Enter your name"<<endl;
      cin>>name;
    }


};
class exam: public virtual pg
{
    public :
```

```
    int ecode;
    void display3()
    {
        cout<<"Enter exam code"<<endl;
        cin>>ecode;
    }

};
class external : public student, public exam
{
    public :
    int marks;
    void display4()
    {
        cout<<"Enter your  Level 1 marks"<<endl;
        cin>>marks;
    }
    void check()
   {
       if ((age>17 )&&(marks>65))
       {
          cout<<"\nAllowed for Level 2 Examination\n";
       }
       else
        cout<<"Start looking  for backups";
   }
};
int main()
{
    external e;
    cout<<"Enter your valid information"<<endl;
    e.display1();
    e.display2();
    e.display3();
    e.display4();
    e.check();
    return 0;
}
```

**Output:**

```
Enter your valid information
Enter your age
```

```
20
Enter your name
Yash
Enter exam code
23099
Enter your internal marks
68

Allowed for externals
```

**Code Debriefing**

- In this code, we established a relationship between two classes using virtual inheritance.
- This was needed to resolve the issue with multiple inheritance.
- Here, only the child class will now be able to inherit and modify properties from the parent class, as all the members have the same access that is declared in the base class.
- The objects of the child class can access all the members, except the private ones of the base or parent class.

**5.** Hierarchical Inheritance

*Hierarchical inheritance* is when a single base class inherits multiple derived classes. This inheritance has a tree-like structure, since every class acts as a base class for one or more child classes. The syntax for hierarchical inheritance is as follows:

```
class A
{
    // class body
};
class B : accessMode  A
{
    // class body
};
class C : accessMode A
{
    // class body
};
class D : accessMode B
{
    // class body
};
```

```
class E : accessMode C
{
    // class body
};
```

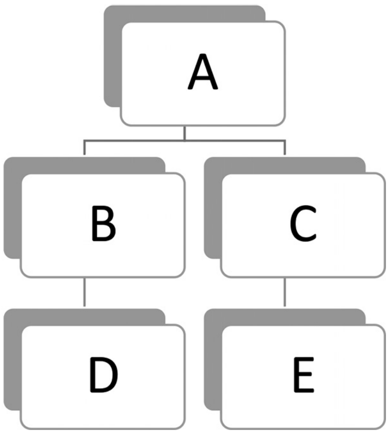


***Figure 6.6*** Hierarchical inheritance

## Code: Working with Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class f1Car
{
    protected :
        string team;
    public:
    void getdata()
    {
        cout<<"Enter team name"<<endl;
        cin>>team;
    }
    void showdata()
    {
      cout<<"Team Name : "<<team<<endl;
    }
};
class engine : public f1Car
{
  protected :
    int LmotrNO;
  public:
  void getdata()
  {
      f1Car :: getdata();
      cout<<"Enter light motor number: ";
      cin>>LmotrNO;
  }
  void showdata()
  {
    f1Car:: showdata();
    cout<<"LMotor No = "<<LmotrNO<<endl;
  }


};
class hyrdrolic : public f1Car
{
  protected:
    float price;
  public :
    void getdata()
  {
      f1Car :: getdata();
      cout<<"Enter heavy motor price: ";
      cin>>price;
  }
  void showdata()
  {
    f1Car :: showdata();
    cout<<"HMotor Price = "<<price<<endl;
  }


};
class fuel: public engine
{
  protected :
  int capacty;
  public:
  void getdata()
  {
     engine :: getdata();
      cout<<"Enter Gear motor capacity: ";
      cin>>capacty;
  }
```

```
    void showdata()
    {
     engine :: showdata();
      cout<<"GMotor Capacity = "<<capacty<<endl;;
    }


  };
  class nofuel : public engine
  {
    protected :
    int capacty;
    public:
    void getdata()
    {
       engine :: getdata();
        cout<<"Enter Non Gear motor capacity: ";
        cin>>capacty;
    }
    void showdata()
    {
     engine :: showdata();
      cout<<"NonGMotor Capacity = "<<capacty<<endl;;
    }

  };
  class race : public hyrdrolic
  {
    protected :
    int passNo;
    public:
    void getdata()
    {
        hyrdrolic :: getdata();
        cout<<"Enter passenger capacity: ";
        cin>>passNo;
    }
    void showdata()
    {
      hyrdrolic:: showdata();
      cout<<"Passenger Capacity = "<<passNo<<endl;;
    }
```

```
};
class sprint : public hyrdrolic
{
  protected :
    int maxLoad;
  public:
  void getdata()
  {
      hyrdrolic :: getdata();
      cout<<"Enter max goods load: ";
      cin>>maxLoad;
  }
  void showdata()
  {
    hyrdrolic :: showdata();
    cout<<"GMotor Capacity = "<<maxLoad<<endl;;
  }

};
int main()
{
  fuel f1;
  race r1;
  cout<<"Enter details of vehicle: "<<endl;
  f1.getdata();
  r1.getdata();
  cout<<"Vehicle Specifications are: "<<endl;
  f1.showdata();
  r1.showdata();
}
```

**Output:**

```
Enter details of vehicle:
Enter team name
Redbull
Enter light motor number: 27789
Emter Gear motor capacity: 4900
Enter team name
Redbull
Enter passenger capacity: 1
Vehicle Specifications are:
Team Name : Redbull
```

```
LMotor No = 27789
GMotor Capacity = 4900
Team Name : Redbull
HMotor Price = 450000
Passenger Capacity = 1
```

**6.** Hybrid Inheritance

Lastly, *hybrid inheritance* is a mixture of all the types of inheritance we have learned.

**Code: Working with Hybrid Inheritance**

```
#include <iostream>
using namespace std;
class pg
{
   public:
   int age;
   void display1()
   {
      cout<<"Enter your age"<<endl;
      cin>>age;
   }


};

class student : public pg
{
    public :
    string name;
    void display2()
    {
        cout<<"Enter your name"<<endl;
      cin>>name;
    }


};
class exam
{
    public :
    int ecode;
    void display3()
    {

        cout<<"Enter exam code"<<endl;
        cin>>ecode;

    }

};
class external : public student, public exam
{
    public :
    int marks;
    void display4()
    {
        cout<<"Enter your Level1 marks"<<endl;
        cin>>marks;
    }
    void check()
   {
       if ((age>17 )&&(marks>65))
       {
           cout<<"\nAllowed for Level 2 Examination :)\n";
       }
       else
        cout<<"Start looking for backups :(( \n";
   }


};

int main()
{
    external e;
    cout<<"Enter your valid information"<<endl;
    e.display1();
    e.display2();
    e.display3();
    e.display4();
    e.check();
    return 0;
}
```

**Output:**

```
Enter your valid information
Enter your age
20
Enter your name
Mick
Enter exam code
203
Enter your Level 1 marks
24
Start looking for backups
```

## 6.4 Constructor Calling

In the previous chapter, we discussed class making and methods, as well as special methods like constructors and their various types. There, we dealt with only a single class, but with inheritance, there will always be more than one class. How does constructor calling work? Let's take a look.

**Remember**

- The base class constructor is always called first, irrespective of the object made.
- Derived class object creation results in initializing all of its members, but it has inherited members, too. Hence, the base class constructor is called first to enable all members to initialize in the derived class.
- In the case of multiple inheritance, the base class order of placement will decide which constructor is called.
- Destructors are always called in the opposite order of the constructors.

The syntax for calling a constructor is as follows:

```
class A
{
    //class body
  A();
};
class B : accessMode  A
{
    // class body
   B();
};
class C : accessMode B
{
     // class body
    C();

};

C object;
```

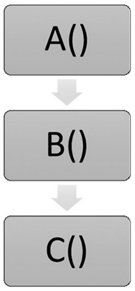


***Figure 6.7*** Constructor order

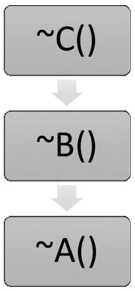


***Figure 6.8*** Destructor order

## 6.5 Implementing Inheritance

Inheritance is an important object-oriented programing concept that you use with classes. A few of the advantages of inheritance are as follows:

1. *Code reusability*: As the relationship between parent and child classes allows for the sharing of methods, we do not need to write the same code body again and again.
2. *Structure:* It makes code easier to understand, as methods do not repeat and the child objects can access all the inherited methods.

3. *Efficiency:* Inheritance saves a lot of time and energy in fetching data from one class to obtain the result.
4. *Extensibility*: Classes (as well members) are able to extend their features and functionalities through derived or child classes.

The following example of inheritance highlights its importance and ease of use.

**Code: Programming with Inheritance**

```
#include <iostream>
using namespace std;
class person
{
   public:
   int age;
   void display1()
   {
      cout<<"Enter your age"<<endl;
      cin>>age;
   }

};
class student : public person
{
    public :
    int ugYear;
    void display2()
    {
        cout<<"Enter your Under Graduate passing Year"<<endl;
        cin>>ugYear;
    }

};
class exam: public student
{
    public :
    void check()
    {
        if((age>18) && (ugYear== 2021 || ugYear== 2022))
            cout<<"You Can appear for CAT 2022"<<endl;
        else
           cout<<"Sorry you Cannot appear for CAT 2022"<<endl;
    }

};

int main()
{
    exam e;
    cout<<"Enter  valid information"<<endl;
    e.display1();
    e.display2();
    e.check();
    return 0;
}
```

**Output:**

```
Enter valid information
Enter your age
19
Enter your under Graduate passing year
2023
Sorry you cannot  appear for CAT 2022
```

Do your own "code debriefing" of this code and try making your own versions of inheritance in C++.

## Summary

- *Inheritance* in C++ allows a child class to inherit features of its parent class.
- The parent class (or the base class or super class) is a single entity that existed before the child class, and all its children inherit properties from it.
- The child class (or the derived class or sub class) is made out of certain features of the base class.
- Data members declared public can be accessed by any class, irrespective of the relationship its class has with other classes.
- Data members declared private have limited access available to only the base, friend, and derived classes.
- Data members declared protected can be accessed by base class members and friend classes.

- Inheritance in C++ can be done in various forms and combinations. There are several types of inheritance that can be implemented: single, multilevel, multiple, virtual, hierarchical, and hybrid.
- The base class constructor is always called first irrespective of the number of objects made.
- Destructors are called in the opposite order of the constructors.
- Inheritance advantages are as follows: code reusability, structure, efficiency, and extensibility.

## Exercises

### Theory Questions

1. What is inheritance? Explain the need for inheritance with suitable examples.
2. What are the differences between the access specifiers private and protected?
3. What are base and derived classes?
4. Explain the syntax for declaring the derived class. Draw an access privilege diagram for the members of a base and a derived class.
5. What are the different forms of inheritance supported by C++? Explain them with an example.
6. Can a base class access members of a derived class? Give reasons.
7. What is accessibilty mode? What are the different inheritance visibility modes supported by C++?
8. What are the differences between inheriting a class with public and private visibility modes?

### Practical Questions

1. Write a program in C++ to create a base class called Stack and a derived class called MyStack. Write a program to use these classes for manipulating objects.
2. Write a program in C++ to declare two classes named Window and Door. Derive a new class called House from those two classes. The Window and Door base classes must have attributes that reflect a happy

home. All classes must have interface functions, such as overloaded stream operator functions for reading and displaying attributes. Write an interactive program that can be used to model this relationship.

3. Write a program in C++ to depict multilevel inheritance using real life entities.
4. Write a program in C++ to create a relation between grandparent and grandchild classes with overriding methods in both.
5. Write a program in C++ to show access to private, public, and protected classes using inheritance.
6. Write a program in C++ to depict a constructor's purpose in single inheritance.
7. Write a program in C++ to depict a constructor's purpose in multilevel inheritance.
8. Write a program in C++ to depict a constructor's purpose in multiple inheritance.
9. Write a program in C++ to depict a constructor's purpose in hierarchical inheritance.
10. Write a program in C++ to depict a constructor's purpose in virtual inheritance.
11. Write a program in C++ to depict a constructor's purpose in hybrid inheritance.

**MCQ-Based**

1. What is inheritance in C++?
   - **a.** Wrapping of data into a single class
   - **b.** Deriving new classes from existing classes
   - **c.** Overloading of classes
   - **d.** Classes with the same names
2. Which of the following statement is correct about virtual inheritance?
   - **a.** It is a technique to ensure that a private member of a base class can be accessed.
   - **b.** It is a technique to optimize multiple inheritances.

   **c.** It is a technique to avoid the multiple inheritances of the classes.

   **d.** It is a technique to avoid creating multiple copies of the base class code for the derived or child classes.

3. If a class is derived privately from a base class, then

   **a.** no members of the base class are inherited.

   **b.** all members are accessible by the derived class.

   **c.** all the members are inherited by the class, but are hidden and cannot be accessed.

   **d.** no derivation of the class gives an error.

4. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A
{ float d;
   public:
       virtual void func(){
              cout<<"Hello this is class A\n";
       }   };
class B: public A
{
    int a = 15;
   public:
       void func(){
              cout<<"Hello this is class B\n";
       }
};
int main(int argc, char const *argv[])
{
       A *a = new A();
       B b;
       a = &b;
       a->func();
       return 0;
}
```

   **a.** Hello this is class A

   **b.** Hello this is class B

**c.** Error

**d.** Segmentation Fault

**5.** Write the output of the following code.

```
#include <iostream>
#include <string>
using namespace std;
class A
{
       float d;
     public:
       virtual void func(){
              cout<<"Hello this is class A\n";
       }
};
class B: public A
{
       int a = 15;
   public:
       void func(){
              cout<<"Hello this is class B\n";
       }
};
int main(int argc, char const *argv[])
{
       A *a;
       a->func();
       return 0;
}
```

**a.** Hello this is class A

**b.** Hello this is class B

**c.** Error

**d.** Segmentation Fault

**6.** What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A{
      float d;
```

```
   public:
      virtual void func(){
             cout<<"Hello this is class A\n";
      }
};

class B: public A{
       int a = 15;
public:
       void func(){
              cout<<"Hello this is class B\n";
      }
};

int main(int argc, char const *argv[])
{
      B b;
      b.func();
      return 0;
}
```

**a.** Hello this is class B

**b.** Hello this is class A

**c.** Error

**d.** Segmentation fault

**7.** What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A
{
      float d;
   public:
      A(){
              cout<<"Constructor of class A\n";
      }
};

class B: public A
{
      int a = 15;
```

```
    public:
      B(){
              cout<<"Constructor of class B\n";
      }
};

int main(int argc, char const *argv[])
{
      B b;
      return 0;
}
```

**a.** Constructor of class A Constructor of class B

**b.** Constructor of class A

**c.** Constructor of class B

**d.** Constructor of class B Constructor of class A

**8.** What is the syntax of the inheritance of a class?

**a.** class name

**b.** class name: access specifier

**c.** class name: access specifier class name

**d.** None of the above

**9.** How many types of inheritance are there in C++?

**a.** 2

**b.** 3

**c.** 4

**d.** 6

**10.** What is meant by multiple inheritance?

**a.** Deriving a base class from a derived class

**b.** Deriving a derived class from a base class

**c.** Deriving a derived class from more than one base class

**d.** None of the above

**11.** Which of the following advantages do we lose by using multiple inheritance?

**a.** Dynamic binding

**b.** Polymorphism

**c.** Both a & b

**d.** None of the above

**12.** Which symbol is used to create multiple inheritance?

**a.** Dot

**b.** Comma

**c.** Dollar

**c.** None of the above

**13.** What is inherited from the base class?

**a.** Constructor and its destructor

**b.** Operator=() members

**c.** Friends

**d.** All of the above

**14.** What will be the output of the following program?

**NOTE** *This code includes all the required header files.*

```
     class find {
public:
     void print()  { cout <<" In find"; }
};
     class course : public find {
     public:
     void print() { cout <<" In course"; }
       };
class tech: public course { };
int main(void)
{
  tech t;
  t.print();
  return 0;
}
```

**a.** In find

**b.** In course

**c.** Compiler Error: Ambiguous call to print()

**d.** None of the above

**15.** Assume that an integer takes 4 bytes and there is no alignment in the following classes. Predict the output.

```
#include<iostream>
        using namespace std;
        class base {
             int arr[10];
              };
            class b1: public base { };
           class b2: public base { };
          class derived: public b1, public b2 {};
          int main(void)
{
 cout << sizeof(derived);
  return 0;
}
```

**a.** 40

**b.** 80

**c.** 0

**d.** 4

**16.** What is the output of the following program?

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

class Base {
private:
     int i, j;
public:
    Base(int _i = 0, int _j = 0): i(_i), j(_j) { }
};
```

```
class Derived: public Base {
public:
     void show(){
        cout<<" i = "<<i<<"  j = "<<j;
     }
};
int main(void) {
  Derived d;
  d.show();
  return 0;
}
```

**a.** i = 0 j = 0

**b.** Compiler Error: i and j are private in Base

**c.** Compiler Error: Could not call constructor of Base

**17.** A class serves as a base class for many derived classes; this is called

**a.** polymorphism

**b.** multipath inheritance

**c.** hierarchical inheritance

**d.** none of the above

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (b) | **2.** (d) | **3.** (c) | **4.** (b) | **5.** (d) | **6.** (a) | **7.** (a) | **8.** (c) | **9.** (d) | **10.** (c) |
| **11**. (c) | **12.** (b) | **13.** (d) | **14.** (b) | **15.** (b) | **16.** (b) | **17.** (c) | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

## Websites

- Learn CPP, accessed July 2022, *https://www.learncpp.com*
- Codes Cracker, accessed July 2022, *https://codescracker.com*
- Geeks For Geeks, accessed July 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed July 2022, *https://www.udacity.com*
- Scaler, accessed July 2022, *https://www.scaler.com*
- C Plus Plus, accessed July 2022, *https://cplusplus.com*
- Silly Codes, accessed July 2022, *https://sillycodes.com*

CHAPTER 7

# POLYMORPHISM

## 7.1 Introduction

Another important object-oriented programming concept is polymorphism. We briefly touched on it in the first chapter with the Formula 1 example. Polymorphism provides us with multiple forms of a method with different signatures but the same name. Methods can exist in multiple forms by varying the types of parameters and the number of parameters they take in the signature of a function. For example, in Formula 1 racing, there are 10 teams with 2 drivers each. Both drivers are provided with the same equipment and car specifications. However, the result attained by the drivers and their teams always differs as the car (which would be our programming method. has been modified into completely different styles: an F1 car can exist in different forms.

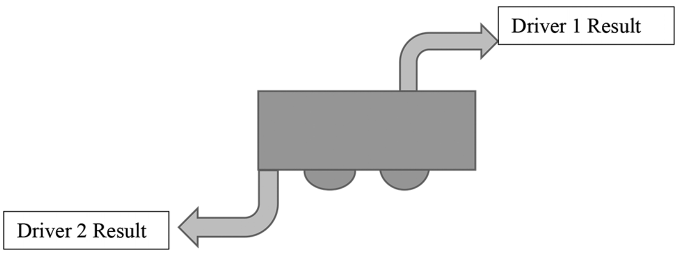


*Figure 7.1* Polymorphism

There are two major types of polymorphism: compile-time polymorphism and runtime polymorphism. (These have subcategories, and the way they are implemented is discussed later.)

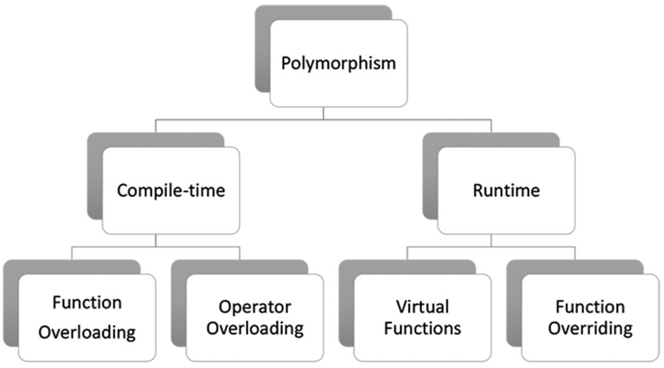


***Figure 7.2*** Polymorphism types

Let's consider an example of display function overloading and how polymorphism works. Here, the syntax is the same as that of the functions we learned before.

## Code: Using Polymorphism and Function Overloading

```
#include <iostream>
using namespace std;
void display(int i)
{
  cout << " Integer   is " << i << endl;
}
void diaplay(double  f)
{
  cout << "Float is " << f << endl;
}
void display(char const *c)
{
  cout << " Character is " << c << endl;
}

int main()
{
    display(11);
    display(11.11);
    display("Eleven");
    return 0;
}
```

**Output:**

```
Integer is 11
Integer is 11
Character is Eleven
```

### Code Debriefing

- This code shows that the display function of our class exists in multiple forms.
- This type of polymorphism is implemented using function overloading.
- The functions all have the same name but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.

### Code: Polymorphism and Addition Methods

```
#include <iostream>
using namespace std;
int add(int num1, int num2) {
    return num1 + num2;
}
double add(float num1, double num2) {
    return num1 + num2;
}
int add(int num1, float num2, int num3) {
    return num1 + num2 + num3;
}
int main() {
    cout << "\nPolymorphism of addition method  " << endl;
    cout << "Addition method 1 = " << add(87, 45) << endl;
    cout << "Addition method 2 = " << add(90.5, 63.6) << endl;
    cout << "Addition method 3 = " << add(51, 5.6, 33) << endl;

    return 0;
}
```

**Output:**

```
Polymorphism of addition method
Addition method 1 = 132
Addition method 2 = 154.1
Addition method 3 = 89
```

**Code Debriefing**

- In this code, the addition function of our class exists in multiple forms.
- This type of polymorphism is implemented using the function overloading of the addition method.
- The functions all have the same name, but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.

## 7.2 Dynamic vs. Static Binding

***Table 7.1*** Dynamic and static binding

| Dynamic Binding | Static Binding |
|---|---|
| Dynamic or late binding tells which procedure will be called at runtime. | *Static* or *early binding* tells which procedure will be called at compile time. |
| The code inside our procedure is not known until execution, hence the name late binding. | The code inside our procedure is known before execution, and both join together at execution time, hence the name *early binding*. |
| It is implemented in C++ using virtual functions. | It is implemented in C++ using normal function calls, function overloading, and operator overloading. |
| The execution of a program is slower. | The execution of a program is faster. |
| Runtime polymorphism is also another name for it. | *Compile-time polymorphism* is also another name for it. |

**Code: Static binding**

```
#include <iostream>
using namespace std;
class person
{
   public:
   int age;
   void display1()
   {
      cout<<"Enter your age"<<endl;
      cin>>age;
   }
};
class student : public person
{
    public :
    int ugYear;
    void display2()
    {
        cout<<"Enter your Undergraduate Year "<<endl;
        cin>>ugYear;
    }


};
class exam: public student
{
    public :
    void check()
    {
        if((age>18) && (ugYear== 2021 || ugYear== 2022))
              cout<<"You can appear for Common Admission Test
2022"<<endl;
        else
            cout<<"Sorry, you cannot appear "<<endl;

    }

};

int main()
{
    exam e;
    cout<<"Enter your valid information"<<endl;
    e.display1();
    e.display2();
    e.check();
    return 0;
}
```

**Output:**

```
Enter your valid information
Enter your age
```

```
21
Enter your Undergraduate Year
2024
Sorry, you cannot appear
```

### Code Debriefing

- This code shows that the display function of our class exists in multiple forms.
- Static binding is implemented here using function overloading.
- The functions all have the same name, but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.
- Here, the compiler gets to know about the code attached to the function call only at compile time.

### Code: Dynamic binding

```
#include <iostream>
using namespace std;
class person
{
   public:
   virtual void display()
   {
      cout<<"Enter your age"<<endl;
   }

};
class student : public person
{
    public :
    void display()//overridden
    {
        cout<<"Enter your  Under Graduaate Year"<<endl;
    }
    };

int main()
{
    person e;
    student s;
    cout<<"Please enter your information "<<endl;
s.display();
e.display();
    return 0;
}
```

**Output:**

```
Please enter your information
Enter your undergraduate Year
Enter your age
```

**Code Debriefing**

- This code shows that the display function of our class exists in multiple forms.
- Static binding is implemented using function overloading.
- The functions all have the same name, but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.
- Here, the compiler gets to know about method attached to the function call only at runtime.

## 7.3 Interface and Implementation

Sometimes, a code block must be redefined again and again due to the needs of the child classes. For this, C++ provides interfaces, which lead to no direct implementation and only declare methods. Let us look at some of the features of an interface:

**a.** No method of implementation is given; they only deal with the declaration part.

**b.** It is the responsibility of the class that implements an interface to implement the methods, as well.

**c.** We cannot define variables inside our interface.

### Code: Working with Interfaces

```
#include <iostream>
using namespace std;
class myInterface
{
public:
virtual void Purely() = 0;
void method1()
{
cout<<"\nParent class method"<<endl;
}
};
class child1: public myInterface
{
public:
void Purely()
{
cout<<"\nParent method redefined"<<endl;
}
};

int main()
{
cout<<"\nInterfaces"<<endl;
child1 c1;
c1.Purely();
c1.method1();
return 0;
}
```

### Output:

```
Interfaces
Parent method redefined
Parent class method
```

### Code Debriefing

- Here, the interface concept is being applied.
- Static binding is implemented using function overloading.
- The method is redefined in the child class as it can inherit and modify properties.

The following are few more terms and their definitions to make the interface concept in C++ clearer.

- **a.** A *virtual function* is a type of function that is declared as well as defined in the parent or base class and redefined in the child class.
- **b.** A *pure virtual function* is a type of function that is only declared but never implemented or redefined.
- **c.** An abstract class (ABC) is a type of class that has only a pure virtual function. This means that it is only declared and never defined, which helps all child classes to inherit and redefine the members as needed. This type of class cannot be instantiated by directly creating an object, but it can be instantiated through implementing an interface. All pure virtual functions of the class need to be overridden or else an error may occur. The syntax for this is as follows:

```
virtual void functionName() = 0;
```

**Code:**

```
#include <iostream>
using namespace std;
class parentc
{
    public :

        virtual int pro()=0;
    protected:
        int avgSal=230000;



};
class PublicChild1 : public parentc
{
  public:
 virtual int pro()
  {
    return (avgSal+9000);
    }
};

class PublicChild2: public parentc
{
  public:
 virtual int pro()
```

```
    {
      return (avgSal+12000);
      }
};

int main()
{
    PublicChild1 obj1;
    PublicChild2 obj2;
    cout << "\nPublic child 1 average salary is "
         << obj1.pro() << endl;
    cout << "\nPublichild2 average Salary is " << obj2.pro()
         << endl;
    cout<< "\n";

    return 0;
}
```

**Output:**

```
Public child 1 average salary is 239000
Publicchild2 average Salary is 242000
```

**Code Debriefing**

- In this code, the virtual function of our class exists in multiple forms.
- While calling the display function, keep in mind which signature calls which function.

**Code:**

```
#include <iostream>
using namespace std;
class ABC
{
public:
virtual void Purely() = 0;
};
class child1: public ABC
{
public:
void Purely()
{
cout<<"\nParent method redefined by child1"<<endl;
}
};
class child2: public ABC
{
public:
void Purely()
{
cout<<"\nParent method redefined by child2"<<endl;
}
};
int main()
{
child1 c1;
child2 c2;
c1.Purely();
c2.Purely();
return 0;
}
```

**Output:**

```
Parent method redefined by child1
Parent method redefined by child2
```

## 7.4 Function Overriding and Overloading

Now we move onto a type of polymorphism we will be using frequently and that is implemented through functions and their different states. (You should practice this concept by redefining the state and type of function you create and observe how the calling of these using objects takes place.)

*Table 7.2* Function overloading and overriding

| Function Overloading | Function Overriding |
|---|---|
| Compile-time polymorphism implementation | Runtime polymorphism implementation |
| No inheritance required | Inheritance is always required |
| Function overloaded with the same name but different signatures | Function overridden with the same name and signatures |
| Parameter type, number, and order can be changed | Parameter type, number, and order need not be changed |
| Function call determined on the basis of signature done at compile time | Function call determined on the basis of signature done at runtime |
| Overloading may occur multiple times | Overriding occurs only once in a C++ program |

The following examples showcase how functions in C++ are overloaded and overridden.

### Code: Working with Overloaded Functions

```
#include <iostream>
using namespace std;
void display(int i)
{
  cout << " Integer  is " << i << endl;
}
void diaplay(double  f)
{
  cout << "Float is " << f << endl;
}
void display(char const *c)
{
  cout << " The number of characters is  " << c << endl;
}

int main()
{
    cout<<"To show function Ooverloading "<<endl;
    display(11);
    display(11.11);
    display("Eleven");
    return 0;
}
```

### Output:

```
To show function overloading
  Integer is 11
  Integer is 11
The number of characters is Eleven
```

### Code Debriefing

- In this code, the display function of our class exists in multiple forms.
- Polymorphism is being implemented using function overloading.
- The functions all have the same name, but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.

### Code: Function Overloading of the Addition Method

```
#include <iostream>
using namespace std;
int add(int num1, int num2) {
    return num1 + num2;
}
double add(float num1, double num2) {
    return num1 + num2;
}
int add(int num1, float num2, int num3) {
    return num1 + num2 + num3;
}
int main() {

    cout << "Addition method 1 = " << add(87, 45) << endl;
    cout << "Addition method 2 = " << add(90.5, 63.6) << endl;
    cout << "Addition method 3 = " << add(51, 5.6, 33) << endl;

    return 0;
}
```

**Output:**

```
Addition method 1 = 132
Addition method 2 = 154.1
Addition method 3 = 89
```

### Code Debriefing

- This code shows that the addition function of our class exists in multiple forms.
- This type of polymorphism is implemented using function overloading of the addition method.
- The functions all have the same name but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.

### Code:

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

```
class person
{
   public:
   int age;
   void display()
   {
      cout<<"Enter your age"<<endl;
      cin>>age;
   }

};
class student : public person
{
    public :
    int ugYear;
    void display()
    {
        cout<<"Enter your Under GraduateYear"<<endl;
        cin>>ugYear;
    }


};
class exam: public student
{
    public :
    void check()
    {
        if((age>18) && (ugYear== 2021 || ugYear== 2022))
            cout<<"You can appear for Common Admisiion Test
                   2022"<<endl;
        else2
           cout<<"Sorry you cannot appear for Common Admission
                   Test 2022"<<endl;

    }

};

int main()
{
    exam e;
    cout<<"To show function overriding"<<endl;
    cout<<"Please enter your information "<<endl;
    e.display();
    e.check();
    return 0;
}
```

**Output:**

```
To show function overriding
Please enter your information
Enter your Undergraduate Year
2023
Sorry you can not appear for Comman Admission Test 2022
```

**Code Debriefing**

- This code shows that the display function of our class exists in multiple forms.
- This type of polymorphism is implemented using function overriding of the display method.
- It is overridden in the child class, as it is inherited from the base class.
- The functions all have the same name but different signatures. That is, they have different combinations and numbers of parameters.
- While calling the display function, keep in mind which signature calls which function.

## 7.5 Friend and Generic Functions

### 7.5.1 Friend Functions

The friend function is a non-member function of a class that has access as a normal function but is declared with the keyword friend.

**Features of friend functions**

- Declared with the `friend` keyword only once, either with a global scope or in another class.
- It can have any type of access in a class: public, private, or protected.
- Friend functions are called upon without any object or dot operator.
- Objects can be passed as arguments.

- It has access to all data members of a class (including private), even though it is a non-member function.
- Friend relationships cannot be inherited. The syntax for a friend function is as follows:

```
class OurClass
{   //body
    friend  returnType Myfunction(Paramters list…);
};
```

### Code: Using the Friend Function

```
#include <iostream>
using namespace std;
class myclass
{
    friend int myfriend(int a, int b);

};
int myfriend(int a, int b)
    {

        return a+b;
    }

int main()
{
    int a,b;
    cout<<"Enter numbers to add"<<endl;
    cin>>a>>b;
    cout<<"Addition gives "<<myfriend(a,b)<<endl;
    return 0;
}
```

### Output:

```
Enter numbers to add
23
567
Addition gives 590
```

### Code Debriefing

- This code shows how the `friend` function is made.
- This function can be declared inside a class and defined outside the class.

### 7.5.2 Generic Functions

In a situation where a normal function is used more than once, we need parameters on which the function acts to be of different data types. This is made possible using generic functions in C++. These act as a template for different types of parameters to give the desired results.

#### Features of Generic Functions

- Act like macros and body expanded at compile time to receive arguments
- Source code serves as a template and so the code is the same for all functions, but when the different copies of the code are compiled, there are differing results because there are different argument types.
- Class templates can also be used in a similar fashion as generic functions.
- These functions help decrease function overloading.
- Parameters can also be given a default value, if needed.

The syntax for a generic function is as follows:

```
template <typename T> T mytemplate(T paramter1, T parameter2)
{
    //template body
}
```

#### Code: Generic Functions and Swapping Numbers

```
#include <iostream>
using namespace std;
class myclass
{
    public:
    void swapR(int &a,int &b)
    {
       cout<<"Swapping using the call by reference changes what
              is visible "<<endl;
        a= a+b;
        b= a-b;
        a= a-b;
    }
    void swapP(int *a,int *b)
    {
        cout<<"Swapping by using the call by pointers "<<endl;
        *a= *a + *b;
        *b= *a - *b;
        *a= *a - *b;
    }
}c1;
int main()
{
    int a,b;
    cout<<"Enter numbers to swap"<<endl;
    cin>>a>>b;
    cout<<"After swapping A= "<<a<<" B="<<b<<endl;
    c1.swapR(a,b);
    cout<<"After swapping A= "<<a<<" B="<<b<<endl;
    c1.swapP(&a,&b);
    cout<<"After swapping A= "<<a<<" B="<<b<<endl;

    return 0;
}
```

## Output:

```
Enter numbers to swap
23
67
After swapping A= 23 B=27
Swapping using the call by reference changes what is visible
After swapping A= 67 B=23
Swapping by using the call by pointers
```

### Code: Using a Template to Swap Numbers

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

template <class T, class T1>
void swap(T &a,T1 &b)
{
    T temp=a;
    a=b;
    b=temp;
}

int main()
{
    int x;
    float y;
    cout<<"Enter the value of x and y "<<endl;
    cin>>x>>y;
    cout<<"Before Swapping";
    cout<<"\nx1="<<x<<"\ty1="<<y;
    swap(x,y);
    cout<<"\nAfter Swapping";
    cout<<"\nx1="<<x<<"\ty1="<<y;
    return 0;
}
```

**Output:**

```
Enter the value of x and y
34
78
Before Swapping
x1=34  y1=78
After Swapping
x1=78  y1=34%
```

## 7.6 Namespaces

*Namespaces* in C++ logically organize of all our identifiers (variables, methods, and classes) so they are defined and declared under one name or a space. This creates a logical border between various elements and scope created to access the namespace members. In our program, we can either use already defined namespaces like `std::` or define our own space using the keyword namespace.

### Features of Namespaces

- Namespaces created with the same name but different states or scope can exist in C++.
- No access specifiers are required for namespace declarations.
- The nesting of namespaces is possible (a namespace within another namespace).
- It is not a class, so no semicolon is used in the definition.
- Namespace declarations made will only be visible for a global scope.

The syntax for a namespace is as follows:

```
namespace Ournamespace
{
// namespace body
} Ournamespace: : identifiers_any;
```

### Code: Using Namespace

```
#include <iostream>
using namespace  ;
namespace ourSpace1
{
void nFunc()
{
    cout << "\nFunction of ourSpace1 called" << endl;
}
}
namespace ourSpace2
{
void nFunc()
{
    cout << "\nFunction of ourSpace2 called" << endl;
}
}
int main ()
{
ourSpace1::nFunc();
ourSpace2::nFunc();
return 0;
}
```

### Output:

```
Function of ourSpace1 called
Function of ourSpace called
```

### Code Debriefing

- In this code, the namespaces are declared.
- Two functions are defined inside the namespaces.
- The function `nFunc` is called upon using the scope resolution operator.

## Summary

- Polymorphism allows for multiple forms of a method with different signatures but the same name.
- Methods made can exist in multiple forms by varying in type of parameters and number of parameters they take in the signature of a function.
- Dynamic or late binding tells which procedure will be called at runtime.
- Static or early binding tells which procedure will be called at compile time.
- C++ provides interfaces that lead to no direct implementation and only declare methods.
- A *virtual function* is a type of function that is declared as well as defined in the parent or base class and redefined in the child class.
- A *pure virtual function* is a type of function that is only declared but never implemented or redefined.
- The abstract class (ABC) is a type of class that has only pure virtual functions. This means that it is only declared and never defined; this helps all child classes to inherit and redefine the members as they wish, too.
- Function overloading is done by giving functions the same name but different signatures.
- Function overriding is done by giving functions the same name and signatures.
- A *friend* function is a non-member function of a class that has access as a normal function but is declared with the keyword `friend`.
- Generic functions in C++ act as a template and use different types of parameters to give the desired results.
- *Namespaces* logically organize identifiers (variables, methods, and classes) so they can be defined and declared under one name or a space.

## Exercises

### Theory Questions

1. What is polymorphism in C++? Describe the syntax for using it with examples.
2. What are the different types of polymorphism available in C++?
3. What is the difference between compile time and runtime polymorphism?
4. Discuss the concept of virtual and pure virtual functions in C++?
5. What do is meant by dynamic and static binding?
6. How is runtime polymorphism implemented in C++? Explain with examples.
7. What are abstract classes? Explain few of their features.
8. What is the use of a friend function in C++? Explain a few of its features.
9. What are templates in C++? Explain few of its features and how are they are declared.
10. What are namespaces in C++? Explain few of their features.
11. Draw parallels between abstract and virtual functions in C++.

### Practical Questions

1. Write a program in C++ to demonstrate the use of runtime polymorphism.
2. Write a program in C++ illustrating the use of pure virtual functions.
3. Write a program in C++ to depict the use of the abstract keyword.
4. Write a program in C++ to maintain an ecommerce database using virtual functions.
5. Write a program in C++ to use the concept of templates to find the average of five inputs.
6. Write a program in C++ to depict the use of an abstract base class.
7. Write a program in C++ to implement the concept of namespaces.
8. Write a program in C++ to show a student database using various aspects of polymorphism.

## MCQ-Based

1. Which of the following options correctly explains the concept of polymorphism?

   a. `int func(float);`

      `float func(int, int, char);`

   b. `int func(int);`

      `int func(int);`

   c. `int func(int, int);`

      `float func1(float, float);`

   d. None of the above

2. Compile-time polymorphism in C++ language is

   a. Operator overloading

   b. Function overloading

   c. Function overriding

   d. B Only

   e. A & B

3. If the same message is passed to objects of several different classes and all of those can respond in a different way, what is this feature called?

   a. Inheritance

   b. Overloading

   c. Polymorphism

   d. Overriding

4. Which is the best description of polymorphism?

   a. It is the ability for undefined message/data to be processed in at least one way.

   b. It is the ability for a message/data to be processed in more than one form.

   c. It is the ability for many messages/data to be processed in many ways.

   d. None of the above

5. The language that supports classes but not polymorphism is called what?
   a. child-class-based language
   b. class-based language
   c. object-based language
   d. procedure-oriented language
6. Which language does not support polymorphism but supports classes?
   a. C#
   b. Ada
   c. C++
   d. Java
7. Which type of function shows polymorphism?
   a. Inline function
   b. Virtual function
   c. Undefined functions
   d. Class member functions
8. When using an abstract class or function overloading, which function is supposed to be called first?
   a. Local function
   b. Function with the highest priority in the compiler
   c. Global function
   d. None of the above
9. Which one of the following can't be used for polymorphism?
   a. Member functions overloading
   b. Static member functions
   c. Global member function
   d. Constructor overloading
10. Which one of the following can show polymorphism?
   a. Overloading ||

**b.** Overloading &&

**c.** Overloading <<

**d.** Overloading +=

**11.** Two classes are derived from one base class and redefine a function of the base class, also overloading some operators inside the body of the class. Where should polymorphism be used, in the function or the operator overloading?

**a.** Function overloading only

**b.** Operator overloading only

**c.** Either function overloading or operator overloading because polymorphism can be applied only once in a program

**d.** Both of these are using polymorphism

**12.** Which of the following is not true for polymorphism?

**a.** It is a feature of OOP.

**b.** Ease in the readability of a program

**c.** Helps in redefining the same functionality

**d.** Always increases the overhead of the function definition

**13.** Which class or set of classes can illustrate polymorphism in the following code?

```
abstract class student
{
public : int marks;
calc_grade();
}
class topper:public student
{
public : calc_grade()
{
return 10;
}
};
class average:public student
{
public : calc_grade()
{
return 20;
}
};
class failed{ int marks; };

abstract class student
{
public : int marks;
calc_grade();
}
class topper:public student
{
public : calc_grade()
{
return 10;
}
};
class average:public student
{
public : calc_grade()
{
return 20;
}
};
class failed{ int marks; };
```

**a.** Only class student can show polymorphism

**b.** class student, topper and average together can show polymorphism

**c.** Only class student and topper together can show polymorphism

**d.** None of the above

**14.** What is the output of the following code?

```
class student
{
public :
int marks;
void disp()
{
cout<<"its base class";
};
class topper:public student
```

```
{
public :
void disp()
{
cout<<"Its derived class";
}
}
void main() { student s; topper t;
s.disp();
t.disp();
}
```

**a.** its base class and then its derived class

**b.** its base class but not its derived class

**c.** its derived class and then its base class

**d.** None of the above

**15.** Runtime polymorphism is achieved only when a ________ is accessed through a pointer to the base class.

**a.** static function

**b.** real function

**c.** member function

**d.** virtual function

**16.** What is the output of the following program?

```
class education
{
char name[10];
public : disp()
{
cout<<"Its education system";
}
class school:public education
{
public: void disp()
{
cout<<"Its school education system";
}
};
void main()
{
school s;
s.disp();
} }
```

**a.** its school education system

**b.** "its education system and Its school education system

**c.** "its education system"

**d.** None of the above

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (c) | **2.** (e) | **3.** (c) | **4.** (b) | **5.** (c) | **6.** (b) | **7.** (b) | **8.** (b) | **9.** (b) | **10.** (c) |
| **11**. (c) | **12.** (d) | **13.** (b) | **14.** (a) | **15.** (d) | **16.** (a) | | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed July 2022, *https://www.learncpp.com*
- Codes Cracker, accessed July 2022, *https://codescracker.com*
- Geeks For Geeks, accessed July 2022, *https://www.geeksforgeeks.org*
- Programiz, accessed July 2022, *https://www.programiz.com*
- Udacity, accessed July 2022, *https://www.udacity.com*
- Scaler, accessed July 2022, *https://www.scaler.com*
- C Plus Plus, accessed July 2022, *https://cplusplus.com*
- Silly Codes, accessed July 2022, *https://sillycodes.com*

CHAPTER 8

# OPERATOR OVERLOADING

## 8.1 Basics

Polymorphism provides us with multiple forms of a method with different signatures but the same name. Methods can exist in multiple forms by varying the types of parameters and number of parameters they take in the signature of a function. As discussed in the previous chapter, polymorphism encompasses the overloading and overriding of functions and virtual functions. In this chapter, we will consider another type of overloading: operator overloading.

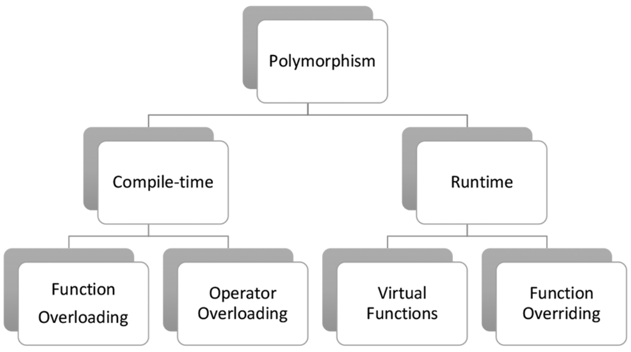


***Figure 8.1*** Polymorphism types

## 8.2 How to Overload an Operator?

*Operator overloading* is a type of compile-time polymorphism in which the basic operators can be overloaded to give a functionality to a user-defined data type. This helps redefine the operators and their operations on operands available in C++ (except for the few as listed below): Even operators can exist in different states of being in C++. Operators that cannot be overloaded because they hold special and complex operability are as follows:

- Scope resolution operator ::
- `Sizeof` operator
- Selector or dot operator .
- Pointer symbol *
- Ternary operator ? :

The syntax for operator overloading is as follows:

```
Return_DataType MyClass : : operator operationName(Parameters
                            list..)
{
     //  function body
}
```

Before loading, check to ensure the following:

- When loading, there should be at least one operand to be operated on in the new user data type.
- No new overloaded operators created can be overloaded again.
- We can even overload using the member method available in a class, but no friend function can be used for this purpose.
- Depending on the type of operator used (whether unary or binary), the member function used will take n+1 operands than in the definition of the operators.

#### Code: Implementing Operator Overloading for ++ Operator

```
#include <iostream>
using namespace std;
class opOverload
{
    int m;
    public:

    opOverload()
    {

    }
   void values()
    {
        cout<<"\nEnter any number\n";
        cin>>m;
    }

    void operator ++()
    {
         m++;

    }
    void display()
    {
       cout<<"\nFirst Number now is "<<m<<endl;
    }

};

int main()
{
    opOverload obj1;
    obj1.values();
    obj1++;
    obj1.display();
    return 0;
}
```

**Output:**

```
Entry any number
33
First number now is 34
```

### Code Debriefing

- In this code, we have created a class with a few methods.
- The `values()` method is used to accept input for data members from the user.

- Here, the unary operator ++ is being overloaded and used with one object (`obj1`).
- This overloading takes no parameter, and the values stored in `obj1` are changed.
- The code now displays the changes made. The overloading was performed successfully.

**Code: Implementing Operator Overloading**

```
#include <iostream>
using namespace std;
class diff
{
    public:
    int m,n;
   void display()
    {
        cout<<"\nEnter any numbers\n";
        cin>>m>>n;
    }
    diff operator == (diff obj1)
    {
        if(m==n)
            cout<<"Equal :)\n";
        else
            cout<<"Not Equal\n";

    }
    diff operator +=(diff obj1)
    {
        m= m+2;
        cout<<"\nFirst number now is "<<m<<endl;
    }

};

int main()
{
    diff obj1,obj2;
    obj1.display();
    obj1==obj1;
    obj1+=obj1;
    return 0;
}
```

**Output:**

```
Entry any numbers
23
56
Not Equal

First number now is 25
```

### Code Debriefing

- In this code, we have created a class with few methods.
- The `display` method is used to take input for the data members from the user.
- Here, the operators == and += are overloaded and used with `obj1`.
- This overloading takes no parameter and the values stored in `obj1` are changed.
- The code now displays the changes made. The overloading was performed successfully.

## 8.3 Overloading Unary Operators

To simply add two numbers (or perform any mathematical operations, such as multiplication), we need the addition operator and the items/values to add. Let's consider an example: in the equation 6+7 = 13, + is the operator and 6 and 7 are the integer values that are our operands. *Unary operators* take only one operand, like -8, which signifies a value of negative 8. The ++ operator is an increment operator acting and modifying one value at a time. In overloading this type of operator by taking only one operand, we made it into polymorphic form. The syntax for overloading a unary operator is as follows:

```
Return_DataType MyClass : : operator operationName(Parameters
                            list..)
{
     //  function body
}
```

### Code: Overloading a Unary Operator

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

```
class unary1
{
    int m;
    public:

    unary1()
    {

    }
   void values()
    {
        cout<<"\nEnter any numbers\n";
        cin>>m;
    }
    void operator ++()
    {
         m++;

    }
    void display()
    {
       cout<<"\nFirst Number now is "<<m<<endl;
    }

};

int main()
{
    unary1 obj1,obj2;
    obj1.values();
    obj1++;
    obj1.display();

    return 0;
}
```

**Output:**

```
Enter any numbers
33
First Number now is 34
```

### Code Debriefing

- In this code, we created a class with a few methods.
- The `values()` method is used to accept input for data members from the user.
- Here, the unary operator ++ is overloaded and used with an object (`obj1`).
- This overloading takes no parameter and the values stored in `obj1` get changed.
- The code now displays the changes made. The overloading was performed successfully.

### Code: Working with Unary Operators

```
#include <iostream>
using namespace std;
class unary2
{
    int m;
    public:

   unary2 ()
    {

    }
   void values()
    {
        cout<<"\nEnter any number\n";
        cin>>m;
    }

    void operator ++()
    {
         m++;

    }
    void operator --()
    {
         m--;

    }
    void display()
    {
        cout<<"\nYour number now is "<<m<<endl;
    }

};

int main()
{
    unary2 obj1;
    obj1.values();
    ++obj1;
    obj1.display();
    --obj1;
    obj1.display();

    return 0;
}
```

**Output:**

```
Enter any number
23
Your number now is 24
Your number now is 23
```

**Code Debriefing**

- In this code, we created a class with few methods.
- The `values()` method is used to accept input for data members from the user.
- Here, the unary operators -- and ++ are being overloaded and used with `obj1`.
- This overloading takes no parameter, and the values stored in `obj1` are changed.
- The code now displays the changes made. The overloading was performed successfully.

## 8.4 Overloading Binary Operators

The second type of operators we have are binary operators, which use two operands. For example, the insertion (<<) and extraction (>>) operators are binary operators. In overloading this type of operator, we take two operands and make them into polymorphic form. The syntax for binary operators is as follows:

```
Return_DataType MyClass : : operator operationName(Parameters
                            list..)
{
     //  function body
}
```

### Code: Overloading a Binary Operator

```
#include <iostream>
using namespace std;
class myclass
{
    public:
    int m,n;
   void  display()
    {
        cout<<"\nEnter any two numbers\n";
        cin>>m>>n;
    }
    int  operator - ()
    {
        return m-n;
    }
};

int main()
{
    myclass obj;
    int r;
    obj.display();
   r=  -obj;
    cout<<"Difference is "<<r <<endl;
    return 0;
}
```

### Output:

```
Enter any two numbers
23
67
Difference is -44
```

### Code Debriefing

In this code, we created a class with a few methods.

- The `values()` method is used to accept input for the data members from the user.
- Here, the binary operator - is overloaded and used with `obj`.
- This overloading takes no parameter and returns with our result. The values stored in the object (`obj`) are changed.
- The code now displays the changes made. The overloading was performed successfully.

### Code: Working with Binary Operators

```
#include <iostream>
using namespace std;
class myclass
{
   private:
      int m;
   public:

       void values()
    {
        cout<<"\nEnter any number\n";
        cin>>m;
    }
       myclass operator *(myclass &obj)
        {
            myclass t;
            t.m= m* obj.m;
           return t;
       }

    void display()
    {
       cout<<"\nYour Result is "<<m<<endl;
    }
};
int main()
{
    myclass obj1,obj2,r;
    obj1.values();
    obj2.values();
    r= obj1*obj2;
    r.display();
    return 0;
}
```

**Output:**

```
Enter any number
48

Enter any number
122

Your Result is 5856
```

### Code Debriefing

In this code, we created a class with a few methods.

- The `values()` method is used to accept input for data members from the user.
- Here, the binary operator * used to perform multiplication is overloaded and used with `objects`.
- An extra temporary object for our class is created inside the function that stores changed values.
- This overloading takes one parameter and the values stored in the object get changed.
- The code now displays the changes made. The overloading was performed successfully.

### Code: Working with Binary Operators

```
#include <iostream>
#include<string.h>
using namespace std;
class concat
{
    char str[20];
    public:

   void input()
    {
        cout<<"\nEnter your string: ";
        cin.getline(str,20);

    }
    void display()
    {
        cout<<"String: "<<str;
    }
    concat operator +(concat s)
    {
        concat obj;
        strcat(str,s.str);
        strcpy(obj.str,str);
        return obj;
    }
};
int main()
{
concat str1,str2,str3;
    str1.input();
    str2.input();
    str3=str1+str2;
    str3.display();
    return 0;
}
```

**Output:**

```
Enter your string: Hello

Enter your string: learners!
String: Hellolearners!%
```

### Code Debriefing

In this code, we created a class with a few methods.

- The `input()` method is used to accept data members from the user for string data types.
- Here, the binary operator + is overloaded and helps concatenate the entered strings.

- Few of string inbuilt functions are used to overload and change functionalities of the binary operator.
- This overloading takes one parameter, and the values stored in the object are changed.
- The code now displays the changes made. The overloading was performed successfully.

## 8.5 Overloading by Friend Function

Overloading in C++ of binary operators can also be done using a friend function. Everything is the same for binary operator overloading, but the difference lies in using the `friend` keyword. The implementation of this friend function is to be carried outside our class.

### Code: Overloading using a Friend Function

```
#include <iostream>
using namespace std;
class myclass {
public:
    int m, n;
    myclass()
    {
        this->m = 0;
        this->n = 0;
    }
    myclass(int x, int y)
    {
        this->m = x;
        this->n= y;
    }
    friend myclass operator+(myclass&, myclass&);
};

myclass operator+(myclass& obj1, myclass& obj2)
{

    myclass obj3;
    obj3.m = obj1.m + obj2.m;
    obj3.n= obj1.n+obj2.n;
    return obj3;
}
```

```
int main()
{
    myclass obj1(86, 99);
    myclass obj2(108, 245);
    myclass obj3;
    obj3= obj1+obj2;
    cout << "\Sum is" << obj3.m<<endl;
    cout << "Sum is "<< obj3.n<<endl;
    return 0;
}
```

**Output:**

```
Sum is 194
Sum is 344
```

### Code Debriefing

- In this code, we have created a class with a few constructors: one default and the other parameterized.
- In this example, overloading is done using a function declared inside the class and implemented outside the class.
- Here, the binary operator + is overloaded and used with `obj1` and `obj2`. The result is being stored in `obj3`.
- This overloading takes two parameters, and the values stored in the objects are changed.
- The code now displays the changes made. The overloading was performed successfully.

## Summary

- Polymorphism provides us with multiple forms of a method with different signatures but the same name.
- *Operator overloading* is a type of compile-time polymorphism in which the basic operators we use can also be overloaded to give a functionality to a user-defined data type.
- Operator overloading is done with two types of operators: unary and binary.

- No new overloaded operators can be overloaded again.
- The operators that cannot be overloaded are the scope resolution operator (`::`), `sizeof` operator, selector or dot operator (`.`), pointer symbol (`*`), and ternary operator (`?`) because they hold a special and complex operability.
- Operator overloading for binary operators can be done using the friend function, as well.

## Exercises

### Theory Questions

1. What is operator overloading? Explain the importance of operator overloading.
2. List the operators that cannot be overloaded and justify why they cannot be overloaded.
3. What is the operator function? Describe the operator function with syntax and examples.
4. What are the limitations of overloading the unary increment/decrement operator? How are they overcome?
5. Explain the syntax of unary operator overloading. How many arguments are required in the definition of an overloaded unary operator?
6. Explain the syntax of binary operator overloading. How many arguments are required in the definition of an overloaded binary operator?

### Practical Questions

1. Write a program to overload the unary operator, say ++, for incrementing the distance in a GPS system. Describe the working model of an overloaded operator with the same program.
2. Write a program in C++ to overload the unary operator for processing counters. It should support both upward and downward counting. It must also support operators for adding two counters and storing the result in another counter.
3. Write a program in C++ to overload arithmetic operators for manipulating vectors.

4. Write a program in C++ to overload new operators and delete operators to manipulate objects of the `Student class`. The `Student` class must contain data members, such as `char*name`, `int roll_no`, and `int branch`. The overloaded new operators and delete operators must allocate memory for the `Student` class object and its data members.

5. Write a program in C++ to manipulate N objects of the Student class. Overload the subscript operator for bounds checking while accessing it.

**MCQ-Based**

1. Which is the correct statement about operator overloading in C++?

   **a.** Only arithmetic operators can be overloaded.

   **b.** Associativity and precedence of operators does not change.

   **c.** The precedence of operators is changed after overloading.

   **d.** Only non-arithmetic operators can be overloaded.

2. What is operator overloading in C++?

   **a.** Overriding the operator, using the user-defined meaning for a user-defined data type

   **b.** Redefining the way the operator works for user-defined types

   **c.** Ability to provide the operators with some special meaning for a user-defined data type

   **d.** All of the above

3. Which of the following operators cannot be overloaded?

   **a.** * (pointer-to-member operator)

   **b.** :: (scope resolution operator)

   **c.** .* (pointer-to-member operator )

   **d.** All of the above

4. Overloading binary operators using a member function requires_______ argument(s).

   **a.** 2

   **b.** 1

**c.** 0

**d.** 3

**5.** Which of the following operators should be preferred to overloading as a global function rather than a member method?

**a.** Postfix ++

**b.** Comparison Operator

**c.** Insertion Operator <<

**d.** prefix ++

**6.** Which of the following operator functions cannot be global (i.e., it must be a member function)?

**a.** new

**b.** delete

**c.** Conversion operator

**d.** All of the above

**7.** Which of the following is the correct order for the process of operator overloading?

**i.** Define the operator function to implement the required operations.

**ii.** Create a class that defines the data type that is to be used in the overloading operation.

**iii.** Declare the operator function op() in the public part of the class.

**a.** 1-i, 2-ii, 3-iii

**b.** 1-ii, 2-iii, 3-i

**c.** 1-ii, 2-i, 2-iii

**d.** 1-iii, 2-ii, 3-i

**8.** What will be the output of the following C++ code?

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

```
{   static int a;
    public:
       void show()
         {      a++;
                cout<<"a: "<<a<<endl;  }
};
 int A::a = 5;
int main(int argc, char const *argv[])
{
     A a;
     return 0;
}
```

**a.** Error, as a private member a is referenced outside the class

**b.** Segmentation fault

**c.** No output

**d.** Program compiles successfully but gives a runtime error

**9.** What happens when objects s1 and s2 are added?

```
string s1 = "Hello";
string s2 = "World";
string s3 = (s1+s2).substr(5);
```

**a.** Error, because s1+s2 will result in a string and no string has a `substr()` function

**b.** Segmentation fault, as two string cannot be added in C++

**c.** The statements run perfectly.

**d.** Runtime error

**10.** In the case of friend-operator-overloaded functions, what is the maximum number of object arguments a unary-operator-overloaded function can take?

**a.** 1

**b.** 2

**c.** 3

**d.** 0

**11.** In the case of friend-operator-overloaded functions, what is the maximum number of object arguments a binary-operator-overloaded function can take?

**a.** 1

**b.** 2

**c.** 3

**d.** 0

**12.** What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A
{
      static int a;
   public:
      void show()
         {
             a++;
             cout<<"a: "<<a<<endl;
      }
      void operator.()
         {
             cout<<"Objects are added\n";
    }
};
class B
{
     public:
};
int main(int argc, char const *argv[])
{
      A a1, a2;
      return 0;
}
```

**a.** Runtime error

**b.** Runs perfectly

**c.** Segmentation fault

**d.** Compile-time error

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (b) | **2.** (d) | **3.** (d) | **4.** (b) | **5.** (c) | **6.** (c) | **7.** (b) | **8.** (c) | **9.** (c) | **10.** (a) |
| **11**. (b) | **12.** (d) | | | | | | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed August 2022, *https://www.learncpp.com*
- Java Point, accessed August 2022, *https://www.javatpoint.com*
- Codes Cracker, accessed August 2022, *https://codescracker.com*
- Geeks For Geeks, accessed August 2022, *https://www.geeksforgeeks.org*
- C Plus Plus, accessed August 2022, *https://cplusplus.com*
- Silly Codes, accessed August 2022, *https://sillycodes.com*

CHAPTER 9

# STRUCTURE AND UNION

## 9.1 Structure: Declaration and Definition

In C++, classes are one way for user-defined data types to exist. However, we can also use *structure*, which is a user-defined data type that groups different data types together. It can be viewed as an array that stores similar data types under one name. The way structure stores information is different from how a class stores it as by default all members of a structure are possessing public visibility while in a class default visibility if all members are private.

### Features of Structure

- Defined using the keyword `struct` and members access it using the name.
- Memory allocation for a structure in C++ occurs contiguously.
- It has data members and functions that are similar to the variables and functions used elsewhere in C++.
- Data members cannot be initialized inside a structure.
- Initialization of members can be done using curly braces.
- Accessing of members is done using the selector or dot operator.

- A pointer to a structure in C++ uses the following characters: ->.
- Like arrays, structures of structures can be made in C++.

The syntax for a structure is as follows:

```
struct {
   // Declaration of our struct
   //members
};
```

The following programs in C++ use classes and help us observe the difference between a class and a structure.

**Code: Class vs Structure in C++**

```
#include <iostream>
using namespace std;
class Student
{
    public:
    string n;
    Student(string n)
    {
    std::cout << "\nName is " <<n<<endl;
    }
    Student(const Student& s1)
    {
    std::cout << "\nThis is a member of a class\n" <<endl;

    }
};
int main() {

    Student s1("Rio");
    Student s3(s1);

    return 0;
}
```

**Output:**

```
Name is Rio
This is a member of a class
```

### Code Debriefing

- In this code, we created a class with a few methods.
- The values' methods are used to take input for the data members from the user.
- Here, the constructor is overloaded and used with one object.
- The code displays the changes made. The overloading was performed successfully.

### Code: Working with Structure

```
#include <iostream>
using namespace std;
struct OurStructure {
    int x;
    char c;

};
int main()
{

    struct OurStructure  s1;
    s1.x = 85;
    s1.c = 'G';
    cout << "The value is : "<< s1.x << endl;
    cout << "The value is : "<< s1.c << endl;
    cout<<"Size of structure : "<<sizeof(s1)<<endl;
    return 0;
}
```

### Output:

```
The value is : 85
The value is : G
Size of structure : 9
```

### Code Debriefing

- In this code, we create a structure with a few variables.
- The object of our structure is made to access the members.
- Observe that the size of the whole structure becomes different, as in a class.
- The code displays the changes made.

## 9.2 Accessing a Structure

Structure, a user-defined data type in C++, groups different data types together. Accessing of members is done using the structure member operator (.) or using a structure pointer. Here are a few examples depicting how members are accessed and called upon. These examples also show the amount of memory (or bytes) occupied by our structure.

### Code: Accessing a Structure

```
#include <iostream>
using namespace std;
struct OurStructure {
    int x;
    char c;

};
int main()
{
    struct OurStructure  s1;
    struct OurStructure s2 = { 240,'S' };
    s1.x = 835;
    s1.c = 'G';
    cout << "The value is : "<< s1.x << endl;
    cout << "The value is : "<< s1.c << endl;
    cout<<"Size of structure  : "<<sizeof(s1)<<endl;
    cout << "The value is : "<< s2.x << endl;
    cout << "The value is : "<< s2.c << endl;
    cout<<"Size of structure  : "<<sizeof(s2)<<endl;
    return 0;
}
```

**Output:**

```
The value is : 835
The value is : G
Size of structure : 8
The value is : 240
The value is : S
Size of structure : 8

The value is : 835
The value is : G
Size of structure structure : 8
The value is : 240
The value is : S
Size of structure structure : 8
```

### Code Debriefing

- In this code, we create two structures with a few variables. A different syntax is used for structure creation.
- The object of our structure is made to access the members.
- Observe that the size of the whole structure becomes different, as in a class.
- The code displays the changes made.

### Code: Working with Structures

```
#include <iostream>
using namespace std;
struct OurStructure {
    int x;
    char c;

};
int main()
{

    struct OurStructure  s1 = { 240,'S' };
    struct OurStructure* s2 = &s1;
    cout << "The value is : "<< s2->x << endl;
    cout << "The value is : "<< s2->c << endl;
    cout<<"Size of structure : "<<sizeof(s1)<<endl;

    return 0;
}
```

### Output:

```
The value is : 240
The value is : S
Size of structure : 8
```

### Code Debriefing

- In this code, we create two structures with a few variables. A different syntax is used for structure creation.
- The object of our structure is made to access the members.

- Observe that the size of the whole structure becomes different, as in a class.
- The code displays the changes made.

## 9.3 Union

Union that is another user-defined data type in C++. *Union* groups various objects and members of different types and bytes together. Structure and union may seem similar because of their syntax, but they vary in their memory allocation to members. A union will give data variable memory space that is equal to the space occupied by the data variable with the largest size of that respective union.

The syntax for a union is as follows:

```
union UnionName
    {//members
    };
```

### Features

- Defined using the keyword `union` and members accessed using under the name
- Memory allocation for a union is determined by the largest member .
- It has data members and functions that are similar to the variables and functions used elsewhere in C++.
- Data members can be initialized and overwritten if the value changes as the same memory space is affected.
- Accessing of members is done using the selector or dot operator.
- A pointer to a structure in C++ is as follows: ->.

### Code: Working with Union

```
#include <iostream>
using namespace std;
using namespace std;
union OurUnion {
    int x;
    char c;

};
```

```
int main()
{

    union OurUnion u1;
    u1.x = 85;
    u1.c = 'H';
    cout << "The value is : "<< u1.x << endl;
    cout << "The value is : "<< u1.c << endl;
    cout<<"Size of union  : "<<sizeof(u1)<<endl;
    return 0;
}
```

**Ouput:**

```
The value is : 72
The value is : H
Size of union : 4
```

### Code Debriefing

- In this code, we create a union with a few variables.
- Then, the object of our union is made to access the members.
- Observe that the size of the whole union changes, as in a structure.
- The code displays the changes made.

### Code:Implementation of a Union data type

```
#include <iostream>
using namespace std;
using namespace std;
union OurUnion {
    int x;
    char c;
    char name[12];
};
int main()
{union OurUnion u1;
    u1.x = 8588;
    u1.c = 'H';
    union OurUnion u2;
    u2.x=  590;
    u2.c= 'S' ;
    cout << "The value is x : "<< u1.x << endl;
    cout << "The value is c : "<< u1.c << endl;
```

```
        cout<<"Size of union  : "<<sizeof(u1)<<endl;
        cout << "The value is x : "<< u2.x << endl;
        cout << "The value is c : "<< u2.c << endl;
        cout<<"Size of union  : "<<sizeof(u2)<<endl;
        return 0;
    }
```

**Output:**:

```
The value is x : 8250
The value is c : H
Size of union : 12
The value is x : 595
The value is c : S
Size of union : 12
```

**Code Debriefing**

- In this code, we create a union with a few variables and assigned values using the dot operator.
- Then, the object of our union is made to access the members.
- Observe that the size of the whole union changes, as in a structure.
- The code displays the changes made.

## 9.4 Differences Between Structure and Union

Now that we have seen these two user-defined data types in C++, let us consider how the methods differ from one another. The following table will help learners determine when and where structure and union can be used in their C++ program.

***Table 9.1*** Structures & Unions Difference Table

| **Structure** | **Union** |
|---|---|
| Structure is a user-defined data type in C++ that groups different data types together. | Union helps us to group various objects and members of different types and bytes together. |
| Memory allocation for a structure in C++ occurs contiguously. | Memory allocation for a union is determined by the largest member . |
| Data members cannot be initialized inside a structure. This can be done by declaring a structure. | Data members can be initialized and overwritten if the value changes, as the same amount of memory space is affected. |
| All data members are assigned a unique memory space in a structure. | All data members share the memory space equal to the member with the largest size. |

| Structure | Union |
|---|---|
| All data members can be initialized for a structure. | Only the first data member can be initialized for a union. |
| All data members can be accessed together for a structure. | Accessing of data members can be done only one at a time. |

## 9.5 Enum in C++

User-defined data types help our C++ program to be more flexible and user friendly. They make it easier for the viewer to understand the language the computer wants to speak. Enumeration (enum) is a unique data type in C++ that allows us to define data types and name elements of our choice to be considered integral constants.

The syntax for enumeration is as follows:

```
enum datatypename{enum values list….. };
```

### Code: Using Enumeration

```
#include <iostream>
using namespace std;
int main()
{
    enum Gender { max, checo=9,lewis, kimi, mick, ocon };

    int i;
    for (i = checo; i <= ocon; i++)
        cout << i << " ";

    return 0;
}
```

**Output:**

```
9 10 11 12 13
```

### Code Debriefing

- In this code, we use `enum` with items.
- The `for` loop traverses the data structure and displays each member as an index number.
- The index is +1, and this can be changed depending on the requirements.

### Code: Working with Enum

```
#include <iostream>
using namespace std;
    enum weekDays { Sunday, Monday, Tuesday=19, Wednesday,
                    Thursday, Friday=23, Saturday };

int main()
{
    enum Gender { max, checo=9,lewis, kimi=18, mick, ocon };
    int i;
    for (i = checo; i <= ocon; i++)
        cout << i << " ";
    weekDays d;
    d= Monday;
    cout << "\n The Day of week is " << d+1<<endl;
    return 0;
}
```

### Ouput:

```
9 10 11 12 13 14 15 16 17 18 19 20
The Day of week is 2
```

### Code Debriefing

- In this code, we use `enum` with items.
- The `for` loop traverses the data structure and displays each member as an index number.
- The index is +1 and, this can be changed depending on the requirements.

### Features of Enumeration

- Defined using the keyword `enum` and elements are accessed using the name
- Memory allocation for all members of `enum` is the same.
- It has data elements and a default position or value assigned that is similar to an array.
- We can change the value of the elements, and the elements following take up the value as defined.

- Accessing of elements can be done by comparing the switch case, as they are integral constants.
- `Enum` elements are called *enumerators* and help simplify the complex body of code.

## Summary

- *Structure* is a user-defined data type in C++ that groups different data types together.
- *Union* helps us to group various objects and members of different types and bytes together.
- Memory allocation for a structure in C++ occurs contiguously.
- In a union, all data members share the memory space equal to the size of the largest member.
- All data members can be initialized for a structure.
- Only the first data member can be initialized for a union.
- `enum` or enumeration is a unique data type in C++ that allows us to define data types and named elements of our choice by assigning names to integer constants.
- `Enum` elements are called *enumerators*.

## Exercises

### Theory Questions

1. What are structures? Justify their need with an example.
2. Why are structures are called heterogeneous data types?
3. Explain storage organization of structure variables.
4. Write a short note on passing structure type variables to a function, and the suitability of different parameter passing schemes in different situations.
5. What are unions? Write a program to illustrate the use of a union.
6. What are the differences between structures and unions?
7. Define enum and its purpose.

**Practical Questions**

1. Write a program in C++ for processing admission reports. Use a structure that has elements representing information such as roll number, name, date of birth (use a nested structure), and branch allotted. The functions processing the members of a structure must be a part of a structure. The format of the report is as follows:

| Roll.no. | Name | Date of Birth | Branch Allotted |
|---|---|---|---|
| XX | xxx | dd/mm/yy | xxxxxxxxxxx |

2. Write a program in C++ that processes the date of birth using structures. Include the ability to process multiple students' dates of birth.

3. Write a program in C++ to process complex numbers. It has to perform addition, subtraction, multiplication, and division of complex numbers. Print the results in x+ iy form.

4. Consider the following structure declaration:

```
struct institution {
struct teacher{
int empl_no;
 char name[20];
};
struct student {
int roll_no;
char name[15];
};
};
```

What are the values for sizeof(institution), sizeof(teacher), and sizeof(student)?

**MCQ-Based**

1. What is the correct output of the given code?

```
#include <iostream>
#include <math.h>
using namespace std;
struct st {
    int A = NULL;
    int B = abs(EOF + EOF);
} S;
int main()
{
    cout << S.A << " " << S.B;
    return 0;
}
```

**a.** 0 2

**b.** 0 0

**c.** 2 0

**d.** Error

**2.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
typedef struct{
    int A = 10;
    int B = 20;
} S;
int main()
{
    cout << S.A << " " << S.B;
    return 0;
}
```

**a.** Error

**b.** 10 20

**c.** 10 0

**d.** 0 0

**3.** What is the correct output of the given code?

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

typedef struct{
    int A;
    char* STR;
} S;

int main()
{
    S ob = { 10, "india" };

    S* ptr;
    ptr = &ob;

    cout << ptr->A << " " << ptr->STR;
    return 0;
}
```

**a.** 10 india

**b.** Blank output

**c.** Garbage value

**d.** India 10

**4.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    typedef struct
    {
        int A;
        char* STR;
    } S;
    S ob = { 10, "india" };
    S* ptr;
    ptr = &ob;
    cout << ptr->A << " " << ptr->STR;
    return 0;
}
```

**a.** 10 in

**b.** 10 d

**c.** 10 india

**d.** 10 n

**5.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
struct st {
    int A;
    char CH;
};
int main()
{
    struct st s[] = { { 10, 'A' }, { 20, 'B' } };
    int X, Y;
    X = s[0].A + s[0].CH;
    Y = s.A + s.CH;
    cout << (X * Y);
    return 0;
}
```

**a.** 6450

**b.** 1020

**c.** 9765

**d.** Error

**6.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
struct st1 {
    int A = 10;
    struct st2 {
        char ch = 'A';
    } S;
} SS;
int main()
{
    struct st1* PTR;
    int X = 0;
    X = PTR->A + PTR->S.ch;
    cout << X;
    return 0;
}
```

**a.** 6450

**b.** Garbage value

**c.** Error

**d.** 0

**7.** What is the correct output of the given code ?

```
#include <iostream>
using namespace std;
struct st1 {
int A = 10;
 struct st2 {
 char ch = 'A';
    } S;

} SS;

int main()
{
    struct st1* PTR = &SS;
    int X = 0;
    X = PTR->A + PTR->S.ch;
    cout << X;
    return 0;
}
```

**a.** 65

**b.** 75

**c.** 97

**d.** 107

**8.** What will be the output of the following code?

```
#include <iostream>
#include <string.h>
using namespace std;
int main() {
int student{
        int num;
        char name[25];
    }
    student stu;
    stu.num = 123;
    strcpy(stu.num, "john");
    cout << stu.num << endl;
    cout << stu.name << endl;

    return 0;
}
```

**a.** 123
john

**b.** john
john

**c.** compile time error

**d.** runtime error

**9.** What will be the output of the following code?

```
#include <iostream>
using namespace std;
struct Time{
    int hours, minutes, seconds;
}
int toSeconds(Time now);
int main() {
    Time t;
    t.hours = 5;
    t.minutes = 30;
    t.seconds = 45;
    cout << "Total seconds: " << toSeconds(t) << endl;
    return 0;
}
int toSeconds (Time now){
    return 3600 * now.hours + 60 * now.minutes + now.seconds;

}
```

**a.** 19845

**b.** 20000

**c.** 15000

**d.** 19844

**10.** What will be the output of the following code?

```
#include <iostream>
using namespace std;
int main() {
    struct ShoeType{
        string style;
        double  price;
    };
    ShoeType shoe1, shoe2;
    shoe1.style = "Adidas";
    shoe1.price = 9.99;
    cout << shoe1.style << "$" << shoe1.price;
    shoe2 = shoe1;
    shoe2.price = shoe2.price / 9;
    cout << shoe2.style << "$" << shoe2.price;
    return 0;
}
```

**a.** Adidas $ 9.99Adidas $ 1.11

**b.** Adidas $ 9.99Adidas $ 9.11

**c.** Adidas $ 9.99Adidas $ 11.11

**d.** Adidas $ 11.11Adidas $ 11.11

**11.** Which of the following is a properly defined structure?

**a.** `struct {int a;}`

**b.** `struct a_struct {int a;}`

**c.** `struct a_struct int a;`

**d.** `struct a_struct {int a;};`

**12.** Which of the following accesses a variable in structure *b?

**a.** `b->var;`

**b.** `b.var;`

**c.** `b-var;`

**d.** `b>var;`

**13.** Which of the following are themselves a collection of different data types?

**a.** Strings

**b.** Structures

**c.** Characters

**d.** None of the above

**14.** Which operator connects the structure name to its member name?

**a.** -

**b.** ->

**c.** .

**d.** Both . and ->

**15.** Which of the following cannot be a structure member?

**a.** Function

**b.** Array

**c.** Structure

**d.** None of the above

**16.** What is the correct syntax to declare a function foo() that receives an array of structures in a function?

**a.** `void foo(struct *var);`

**b.** `void foo(struct *var[]);`

**c.** `void foo(struct var);`

**d.** None of the above

**17.** Union differs from structure in the following way:

**a.** All members are used at the same time.

**b.** Only one member can be used at a time.

**c.** Unions cannot have more members.

**d.** Unions initialize all members as a structure.

**18.** The data elements in the structure are also known as what?

**a.** objects

**b.** members

**c.** data

**d.** objects and data

**19.** What will be used when terminating a structure?

**a.** `:`

**b.** `}`

**c.** `;`

**d.** `;;`

**20.** What will happen when the structure is declared?

**a.** It will not allocate any memory.

**b.** It will allocate the memory.

**c.** It will be declared and initialized.

**d.** It will be declared.

**21.** The declaration of the structure is also called a

**a.** structure creator

**b.** structure signifier

**c.** structure specifier

**d.** structure creator and signifier

**22.** The size of the following union, where int occupies 4 bytes of memory is ______________.

```
union demo
{
  float x;
  int y;
  char z[10];
};
```

**a.** 8 bytes

**b.** 4 bytes

**c.** 10 bytes

**d.** 18 bytes

**23.** Members of a union are accessed as ______________.

**a.** union-name.member

**b.** union-pointer->member

**c.** Both a & b

**d.** None of the above

**24.** It is not possible to create an array of pointers to structures.

**a.** TRUE

**b.** FALSE

**c.** May be possible in some situations

**d.** Cannot give an answer

**25.** Which of the following statements is true?

**a.** The user has to explicitly define the numeric value of enumerations.

**b.** The user has control over the size of enumeration variables.

**c.** Enumeration can have an effect local to the block, if desired.

**d.** Enumerations have a global effect throughout the file.

**26.** `Sizeof a union` gives the size of the longest element in the union.

**a.** Yes

**b.** No

**c.** May be possible in some situations

**d.** Cannot give an answer

**27.** What is the similarity between a structure, union, and enumeration?

**a.** All of them let you define new values.

**b.** All of them let you define new data types.

**c.** All of them let you define new pointers.

**d.** All of them let you define new structures.

**28.** Which of the following share a similarity in syntax?

1. Union
2. Structure
3. Arrays
4. Pointers

- **a.** 3 and 4
- **b.** 1 and 2
- **c.** 2 and 3
- **d.** All of the above

**29.** The size of a union is determined by size of the

- **a.** first member in the union
- **b.** last member in the union
- **c.** sum of the sizes of all members
- **d.** biggest member in the union

**30.** Which operator connects the structure name to its member name?

- **a.** -
- **b.** .
- **c.** Both (b and (c)
- **d.** None of the above

**31.** How can you free the allocated memory?

- **a.** remove(var-name);
- **b.** free(var-name);
- **c.** delete(var-name);
- **d.** dalloc(var-name);

**32.** Which of the following accesses a variable in structure b?

- **a.** `b->var;`
- **b.** `b.var;`
- **c.** `b-var;`
- **d.** `b>var;`

**33.** Which of the following accesses a variable in structure *b?

- **a.** `b->var;`
- **b.** `b.var;`

**c.** `b-var;`

**d.** `b>var;`

**34.** Which of the following is a properly defined structure?

**a.** `struct {int a;}`

**b.** `struct a_struct {int a;}`

**c.** `struct a_struct int a;`

**d.** `struct a_struct {int a;};`

**35.** Which properly declares a variable of structure foo?

**a.** `struct foo;`

**b.** `struct foo var;`

**c.** `foo;`

**d.** `int foo;`

**36.** What is the output of this program?

```
#include <stdio.h>
struct test {
    int x = 0;
    char y = 'A';
};
int main()
{
    struct test t;
    printf("%d, %c", s.x, s.y);
    return 0;
}
```

**a.** 0

**b.** Error

**c.** garbage value garbage value

**d.** None of the above

**37.** What is the output of this program?

```
#include <stdio.h>
struct test {
    int x;
    char y;
} test;
```

```
int main()
{
    test.x = 10;
    test.y = 'A';
    printf("%d %c", test.x,test.y);
    return 0;
}
```

**a.** 0.416666666666667

**b.** garbage value garbage value

**c.** Compilation Error

**d.** None of the above

**38.** What is the output of this program?

```
#include <stdio.h>
struct result{
  char sub[20];
  int marks;
};
void main()
{
      struct result res[] = { {"Maths",100},
      {"Science",90},
      {"English",85}
      };
    printf("%s ", res.sub);
    printf("%d", (*(res+2)).marks);
}
```

**a.** Maths 100

**b.** Science 85

**c.** Science 90

**d.** Science 100

**39.** What will be the size of the following structure?

```
struct demo{
  int a;
  char b;
  float c;
}
```

**a.** 12

**b.** 8

**c.** 10

**d.** 9

**40.** What is the output of this program?

```
#include <stdio.h>
void main()
{
struct demo{
    char * a;
    int n;
};

struct demo p = {"hello", 2015};
struct demo q = p;
printf("%d", printf("%s",q.a));
}
```

**a.** hello

**b.** 5hello

**c.** hello5

**d.** 6hello

**41.** What is the output of this program?

```
#include <stdio.h>
int main()
{
    union demo {
        int x;
        int y;
    };

    union demo a = 100;
    printf("%d %d",a.x,a.y);
}
```

**a.** 100 0

**b.** 100 100

**c.** 0 0

**d.** Compilation Error

**42.** What is the output of this program?

```
#include <stdio.h>
int main()
{
    enum days {MON=-1, TUE, WED=4, THU, FRI, SAT};
    printf("%d, %d, %d, %d, %d, %d", MON, TUE, WED, THU, FRI,
           SAT);
    return 0;
}
```

**a.** -1 0 4 5 6 7

**b.** -1 0 1 2 3 4

**c.** 0 1 2 3 4 5

**d.** Error

**43.** What is the output of this program?

```
#include <stdio.h>
int main(){
   struct simp
      {
             int i = 6;
             char city[] = "chicago";
      };
      struct simp s1;
      printf("%d",s1.city);
      printf("%d", s1.i);
    return 0;
}
```

**a.** chicago 6

**b.** Nothing will be displayed

**c.** Runtime Error

**d.** Compilation Error

**44.** What is the output of this program?

```
#include <stdio.h>

     struct
{
     int i;
     float ft;
}decl;
int main(){
    decl.i = 4;
       decl.ft = 7.96623;
       printf("%d %.2f", decl.i, decl.ft);
       return 0;
}
```

**a.** 4 7.97

**b.** 4 7.96623

**c.** Compilation error

**d.** None of the above

**45.** What is the output of this program?

```
void main()
  {
  struct bitfields {
  int bits_1: 2;
  int bits_2: 9;
  int bits_3: 6;
  int bits_4: 1;
  }bit;
  printf("%d", sizeof(bit));
}
```

**a.** 2

**b.** 3

**c.** 4

**d.** 0

**46.** What is the output of this program?

```
#include <stdio.h>
int main(){
    struct leader
       {
       char *lead;
       int born;
       };
       struct leader l1 = {"Adam ", 1931};
       struct leader l2 = l1;
```

```
        printf("%s %d", l2.lead, l1.born);
}
```

**a.** Compilation error

**b.** Garbage value 1931

**c.** Adam 1931

**d.** None of the above

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (a) | **2.** (a) | **3.** (a) | **4.** (b) | **5.** (a) | **6.** (b) | **7.** (b) | **8.** (a) | **9.** (a) | **10.** (a) |
| **11**. (d) | **12.** (a) | **13.** (b) | **14.** (c) | **15.** (a) | **16.** (a) | **17.** (b) | **18.** (b) | **19.** (c) | **20.** (a) |
| **21**. (a) | **22.** (c) | **23.** (c) | **24.** (b) | **25.** (c) | **26.** (a) | **27.** (b) | **28.** (b) | **29.** (d) | **30.** (b) |
| **31**. (b) | **32.** (b) | **33.** (a) | **34.** (d) | **35.** (b) | **36.** (b) | **37.** (b) | **38.** (a) | **39.** (a) | **40.** (c) |
| **41**. (d) | **42.** (a) | **43.** (d) | **44.** (a) | **45.** (b) | **46.** (c) | | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003)

### Websites

- Learn CPP, accessed August 2022, *https://www.learncpp.com*
- Codes Cracker, accessed August 2022, *https://codescracker.com*
- Geeks For Geeks, accessed August 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed August 2022, *https://www.udacity.com*
- Scaler, accessed August 2022, *https://www.scaler.com*
- C Plus Plus, accessed August 2022, *https://cplusplus.com*
- Silly Codes, accessed August 2022, *https://sillycodes.com*

CHAPTER 10

# EXCEPTION HANDLING

## 10.1 Errors and Exceptions

*Exceptions* in the programming world are runtime abnormal conditions that lead to either program termination or faulty execution. *Errors* are illegal statements or functionality the programmer might have used in their program. Errors are beyond repair but exceptions can be handled.

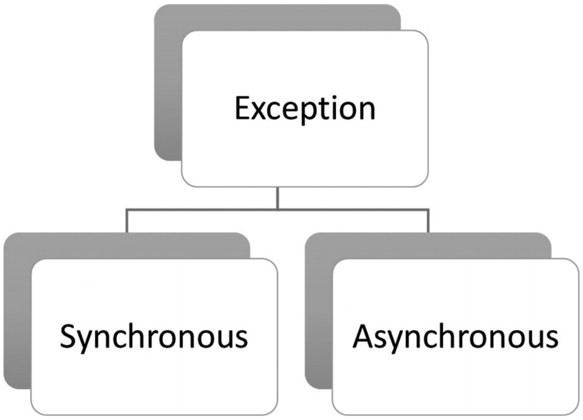


***Figure 10.1*** Types of exceptions

**Exceptions in C++ are broadly divided into two types:**

1. *Synchronous:* This type of exception can only arise from the throw statements. For example, accessing an index element that the array might not have termed an array index generates an out of bounds exception.
2. *Asynchronous:* This type of exception includes keyboard or input interrupts while a program is being executed and disrupts the natural flow of the C++ code. These exceptions are not handled directly but are managed using the signal concept.

**Code: Exception Handling (1)**

```
#include <iostream>
using namespace std;
int main()
{
int x = 0;
int y=10;
cout << "oops wrong \n"<<y/x<<endl;
}
```

**Output:**

```
oops wrong
zsh: floating point exception
```

**Code: Exception Handling (2)**

```
#include <iostream>
using namespace std;
class Parent {

   public:
   int p=60;
   void showP()
    {
        cout<<"\nParent's age  is  " <<p<< endl;

    }

};

class Child : public Parent
{
   public:
   int c=20;
    void showC() {
        cout<< "\nChild's age  is " <<c<< endl;

    }

}//a syntax error will occur as class declaring syntax is not
// being followed
class Grandchild : public Child
{

   public:
   int g=4;
    void showg() {
        cout<< "\nGrandchild's age  is  " <<g<< endl;
        ;
    }

};

int main() {
    Grandchild g1;
    g1.showP();
    g1.showC();
    g1.showg();
    return 0;
}

c
```

**Output:**

```
inherit.cpp:25:2: error: expected ';' after class
}//syntax error will ocur as clas declaring syntax not followed
```

### Code Debriefing

- As you can observe from this code, we did not follow the syntax rules in C++.
- This results in a syntax error message given by the compiler.
- It shows us the line at which error occurred and why the error led to program termination.

## 10.2 Exception Handling

In C++, exceptions are handled using a try-catch block, where we can throw a particular exception we feel our code might face and handle it in the block. Let us now get familiar with important keywords we will be using throughout the exception handling mechanism.

- *Try*: This block is used to throw our exception. We can try for conditions and if the conditions are met (true), we throw an exception. A try block is always followed by a catch block.
- *Catch*: This block is used when an exception is thrown. It gives what happens after catching the exception and managing it with an exception handler. One try block can be followed by multiple catches.
- *Throw*: This keyword simply throws an exception and does nothing much beyond that. It is the responsibility of the try-catch block to handle the exception.

The syntax for the try-catch block is as follows:

```
try {
  // code
  throw exceptionName;
}
catch () {
  // code
}
```

**Code: Handling Exceptions Using Try-Catch Blocks**

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

int main()
{
int x = 0;
int y=10;
//cout << "oops wrong \n"<<y/x<<endl;
try {

    if (y > 9)
    {
        throw y;
    }
}
catch (int y ) {
    cout << "\n The Exception thrown has been Caught \n";
}
return 0;
}
```

**Output:**

```
The Exception thrown has been Caught
```

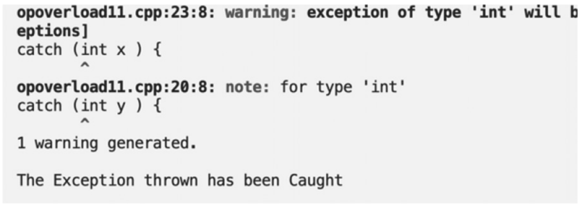
### Code Debriefing

- As you can observe from this code, we use the try-catch block mechanism to handle the exception.
- The `try` block is inside `main`. If the statement is satisfied, we throw our exception in the `try` block.
- The catch block does its work and catches the exception, then prints the message.

### Code: Catching the Exception

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

int main()
{
int x = -9;
int y=10;
//cout << "oops wrong \n"<<y/x<<endl;
try {

    if (y > 9)
    {
        throw y;
    }
    if (x< 0)
    {
        throw x;
    }
}
catch (int y ) {
    cout << "\nThe Exception thrown has been Caught \n";
}
catch (int x ) {
    cout << "\nThe Exception thrown has been Caught \n";
}
return 0;
}
```

**Output:**



```
opoverload11.cpp:23:8: warning: exception of type 'int' will b
eptions]
catch (int x ) {
       ^
opoverload11.cpp:20:8: note: for type 'int'
catch (int y ) {
       ^
1 warning generated.

The Exception thrown has been Caught
```

### Code Debriefing

- As you can observe from this code, we use the try-catch block mechanism to handle exceptions.
- Inside `main`, we have a `try` block. If the statement is satisfied, we throw our exception in the `try` block.
- The `catch` block works in a similar way to a switch case and checks for relevant parameters thrown towards it.
- This code does its work to catch the exception and prints a message.

## 10.3 Various Exceptions

C++ has a list of standard exceptions programmers might see often. A few of them and their functionalities are listed in the following table. They are all part of the std library and are called upon using the `std::exception name`.

***Table 10.1*** List of standard exceptions

| S.No. | Exception | About |
|---|---|---|
| 1. | `exception` | Parent class of all exceptions |
| 2. | `logic_error` | Exception that can be resolved by viewing the code again logically |
| 3. | `out_of_range` | Exception that occurs when a certain value gets beyond the defined scope |
| 4. | `runtime_error` | Exception that cannot be resolved by viewing the code again |

| S.No. | Exception | About |
|---|---|---|
| **5.** | `range_error` | Exception that occurs when we try to operate on out-of-range values |
| **6.** | `overflow_error` | Exception that occurs when values cannot be contained any longer |
| **7.** | `underflow_error` | Exception that occurs where values do not meet the minimum scope |
| **8.** | `length_error` | Exception that can occur where the string values are being operated on |
| **9.** | `domain_error` | Exception that can occur if values are assigned out of the domain |
| **10.** | `bad_alloc` | Exception that can occur while using a new operator for memory |

## 10.4 Custom Exceptions in C++

In C++, if we wish to create our own personalized exceptions, we can do so by inheriting properties from the standard exception class and overriding its method. The following code depicts how the concept of classes and inheriting from predefined classes helps while handling and manufacturing our own exceptions

### Code: Using Custom Exceptions

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

class trial {
};

int main()
{
    try {
        throw trial();
    }

    catch (trial obj) {
        cout << "\nCaught an exception  of a class \n";
    }
}
```

**Output:**

```
Caught an exception of a class
```

### Code Debriefing

- As you can observe from this code, we created our own exception in C++ using the class `trial`.
- Inside `main`, we throw our exception to the class trial constructors within the `try` block.
- The `catch` block does its work and catches the exception, and then it prints the message.
- As our exception is a class, we called its data method using the object.

### Code: Catching a Custom Exception

```
#include <iostream>
#include <exception>
using namespace std;
class ourException : public exception
 {
    public:
   char * func1()
   {
       return "\nUser,Please enter an Exception  ";
   }
};

int main()
{
   try
   {
      throw ourException();
   }
   catch(ourException& oE)
   {
      cout<< "\nOur custom exception has been caught" <<endl;
      cout<< "Calling function "<<oE.func1() <<endl;
   }
   catch(std::exception& oE) {

   }
   return 0;
}
```

**Output:**

```
Our custom exception has been caught
Calling function
User, Please enter an Exception
```

**Code Debriefing**

- As you can observe from this code, we created our own exception by inheriting properties from the `exception` parent class.
- Inside `main`, we throw our exception in the `try` block.
- The `catch` block does its work and catches the exception, and then it prints a message.
- As our exception is a class, we called its data method using the object.

## Summary

- An *exception* in programming is an abnormal condition at runtime that leads to either program termination or faulty execution.
- *Errors* are illegal statements or functionalities the programmer might have used in their program.
- Errors are beyond repair, but exceptions can be handled.
- Exceptions are handled using a `try-catch` block where we can throw a particular exception.
- A `try` block is used to throw an exception. We can check for conditions, and if they are met, the program throws an exception.
- A `catch` block is used when an exception is thrown. What happens after catching the exception occurs in this block.
- A `try` block is always followed by a `catch` block, and one `try` block can be followed by multiple catches.
- There are standard exceptions programmers might face often, and these are part of the `std` library.
- In C++ custom exceptions, we can inherit properties from the standard exception class and override its methods.

## Exercises

### Theory Questions

1. What are exceptions? What are the differences between synchronous and asynchronous exceptions?
2. Explain the exception handling model of C++ with various constructs supported by it.
3. What is the syntax for indicating a list of exceptions that a function can raise? What happens if an unspecified exception is raised?
4. What happens when an exception is raised in a try block having a few constructed objects?
5. What happens when a raised exception is not caught by a catch block?
6. List the ten rules for handling exceptions successfully.

### Practical Questions

1. Write a program to demonstrate the catching of all exceptions.
2. Write a program in C++ to compute the square root of a number. The input value must be tested for validity. If it is negative, should the user-defined function my_sqrt()raise an exception?
3. Write a program in C++ that transfers the control to a user-defined terminate function when a raised exception is uncaught.
4. Write a program in C++ that installs the user-defined unexpected function to handle exceptions.
5. Write a program in C++ that divides two complex numbers. Overload the divide (/) operator. Can this program handle cases such as division-by-zero using exceptions?
6. Write a program in C++ for matrix multiplication. The matrix multiplication function should notify the user if the order of the matrix is invalid using exceptions.
7. Write a program in C++ to add two vectors. Each vector object, an instance of the class Vector, has the dynamic allocation of their data members. Can this program catch exceptions raised by new operators and take corrective actions?

**MCQ-Based**

**1.** What is an exception in a C++ program?

**a.** A problem that arises during the execution of a program

**b.**A problem that arises during compilation

**c.** A syntax error

**d.** A semantic error

**2.** By default, what does a program do when it detects an exception?

**a.** Continues running

**b.** Terminates the program

**c.** Calls other functions of the program

**d.** Removes the exception and tells the programmer about an exception

**3.** Why do we need to handle exceptions?

**a.** To avoid unexpected behavior of a program during runtime

**b.** To let compiler remove all exceptions by itself

**c.** To successfully compile the program

**d.** To get correct output

**4.** Where should we place a catch block of the derived class in a try-catch block?

**a.** Before the catch block of the base class

**b.** After the catch block of the base class

**c.** Anywhere in the sequence of catch blocks

**d.** After all the catch blocks

**5.** What is the syntax for catching any type of exceptions?

**a.** catch (Exception e)

**b.** catch (…)

**c.** catch (Exception ALL)

**d.** catch (ALL)

6. An uncaught exception leads to _________

   a. the termination of the program

   b. the successful execution of the program

   c. no effect on the program

   d. the execution of other functions of the program after it starts

7. What is the header file used for exception handling in C++?

   a. `<cstdlib>`

   b. `<string>`

   c. `<handler>`

   d. `<exception>`

8. The C++ code that causes abnormal termination/behavior of a program should be written under the _________ block.

   a. try

   b. catch

   c. finally

   d. throw

9. Exception handlers are declared with the _________ keyword.

   a. try

   b. catch

   c. throw

   d. finally

10. Which of the following statements are correct about the catch handler?

    i. It must be placed immediately after the try block.

    ii. It can have more than one parameter.

    iii. There must be one, and only one, catch handler for every try block.

    iv. There can be multiple catch handlers for a try block.

    v. General catch handlers can be kept anywhere after the try block.

**a.** i, iv, v

**b.** i, ii, iii

**c.** i, iv

**d.** i, ii

**11.** In a nested try-catch block, if the inner catch block gets executed, then the _________

**a.** program stops immediately

**b.** outer catch block also executes

**c.** compiler jumps to the outer catch block and executes the remaining statements of the `main()` function

**d.** compiler executes the remaining statements of the outer try-catch block and then the `main()` function

**12.** If the inner catch block is unable to handle the exception thrown, then _________

**a.** the compiler looks for the outer try-catch block

**b.** the program stops abnormally

**c.** the compiler will check for the appropriate catch handler of the outer try block

**d.** the compiler will not check for the appropriate catch handler of the outer try block

**13.** In nested try-catch blocks, if both the inner and outer catch blocks are unable to handle the exception thrown, then the _________

**a.** compiler executes only main()

**b.** compiler throws a compile-time error

**c.** program will run without any interruption

**d.** program will be terminated abnormally

**14.** Which function is invoked when an unhandled exception is thrown?

**a.** `stop()`

**b.** `aborted()`

**c.** `terminate()`

**d.** `abandon()`

**15.** How one can restrict a function to throw particular exceptions only?

**a.** By defining multiple try-catch blocks inside a function

**b.** By defining a generic function within a try-catch block

**c.** By defining a function with throw clauses

**d.** Not allowed in C++

**16.** Which function is invoked when we try to throw an exception that is not supported by a function?

**a.** `indeterminate()`

**b.** `unutilized()`

**c.** `unexpected()`

**d.** `unpredicted()`

**17.** The return type of an uncaught exception is _________

**a.** `int`

**b.** `bool`

**c.** `char*`

**d.** `double`

**18.** Which of the following is true about exception handling in C++?

**i.** There is a standard exception class in C++ similar to the exception class in Java.

**ii.** All exceptions are unchecked in C++, i.e., the compiler does not check if the exceptions are caught.

**iii.** In C++, a function can specify the list of exceptions that it can throw using a comma separated list like the following:

```
void fun(int a, char b) throw (Exception1, Exception2, ..)
```

**a.** i, iii

**b.** i, ii, iii

**c.** i, ii

**d.** ii, iii

**19.** Which alternative can replace the throw statement?

**a.** for

**b.** break

**c.** return

**d.** exit

**20.** What are the disadvantages if we use the return keyword to return error codes?

**a.** You have to handle all exceptional cases explicitly.

**b.** Your code size increases dramatically.

**c.** The code becomes more difficult to read.

**d.** All of the above

**21.** In a nested try-catch block, if the inner catch block gets executed, then the _________

**a.** program stops immediately

**b.** outer catch block also executes

**c.** compiler jumps to the outer catch block and executes remaining statements of the `main()` function

**d.** compiler executes the remaining statements of the outer try-catch block and then the `main()` function

**22.** Where are the exceptions handled?

**a.** inside the program

**b.** outside the regular code

**c.** both a & b

**d.** none of the above

**23.** If the inner catch block is unable to handle the exception thrown, then the _________.

**a.** program stops abnormally

**b.** the compiler looks for the outer try-catch block

**c.** the compiler will check for appropriate catch handler of the outer try block

**d.** the compiler will not check for the appropriate catch handler of the outer try block

**24.** Irrespective of the exception occurrence, the catch handler will always get executed.

**a.** True

**b.** False

**25.** Before object-oriented exception handling was practiced, _________.

**a.** no runtime errors occurred

**b.** programmers could not deal with runtime errors

**c.** the most popular error-handling method was to throw an exception

**d.** the most popular error-handling method was to terminate the program

**26.** How do we define user-defined exceptions?

**a.** Inheriting a class functionality

**b.** Overriding a class functionality

**c.** Inheriting and overriding an exception class functionality

**d.** None of the above

**27.** Which type of program is recommended to include in a try block?

**a.** pointer

**b.** const reference

**c.** static memory allocation

**d.** dynamic memory allocation

**28.** We can prevent a function from throwing any exceptions.

**a.** True

**b.** False

**29.** Catch handlers can have multiple parameters.

**a.** True

**b.** False

**30.** What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main()
{
   int var = -12;
   try {
      cout<<"Inside try\n";
      if (var < 0)
      {
        throw var;
        cout<<"After throw\n";
  }
   }
   catch (int var ) {
      cout<<"Exception Caught\n";
   }
   cout<<"After catch\n";
   return 0;
}
```

**a.** Inside try
Exception Caught
After catch

**b.** Inside try
After throw
After catch

**c.** Inside try
Exception Caught
After throw

**d.** Inside try
Exception Caught
After throw
After catch

**31.** What is the output of this program?

```
#include<iostream>
using namespace std;
int main ()
{
try {
int* myarray = new int[1000];
cout << "Allocated";
}
catch (exception& LFC){
cout << "Standard exception: " << LFC.what() << endl;
}
return 0;
}
```

**a.** Error

**b.** Allocated

**c.** Standard exception

**d.** Depends on the memory

**32.** What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catch\n";
        }
    }
    catch (int x)
    {
        cout << "Outer Catch\n";
    }
    return 0;
}
```

**a.** Inner Catch

**b.** Outer Catch

**c.** Inner Catch
Outer Catch

**d.** Error

**33.** What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (char n)
        {
            cout << "Inner Catch\n";
        }
    }
    catch (int x)
    {
        cout << "Outer Catch\n";
    }
    return 0;
}
```

**a.** Inner Catch

**b.** Outer Catch

**c.** Inner Catch
Outer Catch

**d.** Error

**34.** What is the output of the following C++ code?

```
#include <iostream>
#include <typeinfo>
using namespace std;
class A
{
};
int main()
{
    char c; float x;
    if (typeid(c) != typeid(x))
    cout << typeid(c).name() << endl;
    cout << typeid(A).name();
    return 0;
}
```

**a.** c
1A

**b.** x

**c.** Both c & x

**d.** c

**35.** What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
void Division(const double a, const double b);
int main()
{
    double op1=0, op2=10;
    try
    {
        Division(op1, op2);
    }
    catch (const char* Str)
    {
        cout << "\nBad Operator: " << Str;
    }
    return 0;
}
void Division(const double a, const double b)
{
    double res;
    if (b == 0)
        throw "Division by zero not allowed";
    res = a / b;
    cout << res;
}
```

**a.** 0

**b.** Bad operator

**c.** 10

**d.** 15

**36.** What is the output of the following C++ code?

```
#include <stdexcept>
#include <limits>
#include <iostream>
using namespace std;
void MyFunc(char c)
{
    if (c < numeric_limits<char>::max())
        return invalid_argument;
}
int main()
{
    try
    {
        MyFunc(256);
    }
    catch(invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    return 0;
}
```

**a.** 256

**b.** Invalid argument

**c.** Error

**d.** 246

**37.** What is the output of the following C++ code?

```
#include <iostream>
#include <exception>
using namespace std;
class myexc: public exception
{
    virtual const char* what() const throw()
    {
        return "My exception";
    }
} myex;
int main ()
{
    try
    {
        throw myex;
    }
    catch (exception& e)
    {
        cout << e.what() << endl;
    }
    return 0;
}
```

**a.** My

**b.** My exception

**c.** No exception

**d.** Exception

**38.** What is the output of the following C++ code?

```
#include <iostream>
#include <exception>
using namespace std;
int main ()
{
    try
    {
        int* myarray= new int[1000];
        cout << "Allocated";
    }
    catch (exception& e)
    {
        cout << "Standard exception: " << e.what() << endl;
    }
    return 0;
}
```

**a.** Allocated

**b.** Standard exception:

**c.** bad_alloc

**d.** Depends on memory

**39.** What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main()
{
    char* ptr;
    unsigned long int a = (size_t(0) / 3);
    cout << a << endl;
    try
    {
        ptr = new char[size_t(0) / 3];
        delete[ ] ptr;
    }
    catch(bad_alloc &thebadallocation)
    {
        cout << thebadallocation.what() << endl;
    };
    return 0;
}
```

**a.** 0

**b.** 2

**c.** `bad_alloc`

**d.** depends on compiler

**40.** What is the output of the following C++ code?

```
#include <typeinfo>
#include <iostream>
using namespace std;
class shape
{
    public:
    virtual void myvirtualfunc() const {}
};
class mytriangle: public shape
{
    public:
    virtual void myvirtualfunc() const
    {
    };
};
int main()
{
    shape shape_instance;
    shape &ref_shape = shape_instance;
    try
    {
       mytriangle &ref_mytriangle = dynamic_cast<mytriangle&>
        (ref_shape);
    }
    catch (bad_cast)
    {
        cout << "Caught: bad_cast exception\n";
    }
    return 0;
}
```

**a.** Caught standard exception

**b.** No exception arises

**c.** Caught: bad_cast exception

**d.** Caught: cast

**41.** What is the output of the following C++ code?

```
#include <typeinfo>
#include <iostream>
using namespace std;
class Test
{
    public:
    Test();
    virtual ~Test();
};
int main()
{
    Test *ptrvar = NULL;
    try
    {
        cout << typeid(*ptrvar).name() << endl;
    }
    catch (bad_typeid)
    {
        cout << "The object is null" << endl;
    }
    return 0;
}
```

**a.** No exception arises

**b.** The object is null

**c.** Error

**d.** The object is

**42.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0)
            throw 2;

        C = A / B;
    }
    catch (int num) {
        cout << "Error code: " << num << endl;
    }
    return 0;
}
```

**a.** Error code: 2

**b.** Syntax error

**c.** No output

**d.** Garbage value

**43.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;
        C = A / B;
    }
    catch (exception e) {
        cout << "Exception generated" << endl;
    }
    return 0;
}
```

**a.** Exception generated

**b.** Syntax error

**c.** No output

**d.** Program crashed at runtime

**44.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0)
            throw "divide by zero";
        C = A / B;
    }
    catch (char* e) {
        cout << "Exception Received: " << e << endl;
    }
    return 0;
}
```

**a.** Exception received: divide by zero

**b.** Syntax error

**c.** No output

**d.** Program crashed at runtime

**45.** What is the correct output of the given code?

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

int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0)
            throw "divide by zero";
        C = A / B;
    }
    catch (const char* e) {
        cout << "Exception Received: " << e << endl;
    }
    return 0;
}
```

**a.** Exception received: divide by zero

**b.** Syntax error

**c.** No output

**d.** Program crashed at runtime

**46.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;
        if (B == 0)
            throw "Divide by zero";
        C = A / B;
    }
    catch (int num) {
        cout << "Error Code: " << num << endl;
    }
    catch (...) {
        cout << "Exception received" << endl;
    }
    return 0;
}
```

**a.** Exception received

**b.** Divide by zero

**c.** Syntax error

**d.** No output

**47.** What is the correct output of the given code?

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

int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0) {
            Exception E;
            throw E;
        }
        C = A / B;
    }
    catch (exception E) {
        cout << "Exception Received" << endl;
    }

    return 0;
}
```

**a.** Exception received

**b.** Divide by zero

**c.** Syntax error

**d.** No output

**48.** What is the correct output of the given code snippets?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0) {
            exception E;
            throw E;
        }
        C = A / B;
    }
    catch (exception E) {
        cout << "Exception Received" << endl;
    }
    return 0;
}
```

**a.** Exception Received

**b.** Divide by zero

**c.** Syntax error

**d.** No output

**49.** What is the correct output of the given code snippets?

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

int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;

        if (B == 0) {
            bad_exception E;
            throw E;
        }
        C = A / B;
    }
    catch (bad_exception E) {
        cout << "Exception Received" << endl;
    }

    return 0;
}
```

**a.** Exception Received

**b.** Divide by zero

**c.** Syntax error

**d.** No output

**50.** What is the correct output of the given code?

```
#include <iostream>
using namespace std;
int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;
        if (B == 0) {
            bad_exception E;
            throw E;
        }
        C = A / B;
    }
    catch (exception E) {
        cout << "###Exception Received" << endl;
    }
    catch (bad_exception E) {
        cout << "@@@Exception Received" << endl;
    }
    return 0;
}
```

**a.** ###Exception Received

**b.** @@@Exception Received

**c.** Syntax error

**d.** No output

**51.** What is the correct output of the given code?

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

int main()
{
    try {
        int A = 10;
        int B = 0;
        int C;
        if (B == 0) {
            bad_exception E;
            throw E;
        }
        C = A / B;
    }
    catch (bad_exception E) {
        cout << "@@@Exception Received" << endl;
    }
    catch (exception E) {
        cout << "###Exception Received" << endl;
    }
    return 0;
}
```

**a.** ###Exception Received

**b.** @@@Exception Received

**c.** Syntax error

**d.** No output

### MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (a) | **2.** (b) | **3.** (a) | **4.** (a) | **5.** (b) | **6.** (a) | **7.** (d) | **8.** (a) | **9.** (b) | **10.** (c) |
| **11**. (d) | **12.** (c) | **13.** (d) | **14.** (c) | **15.** (c) | **16.** (c) | **17.** (b) | **18.** (b) | **19.** (c) | **20.** (d) |
| **21**. (d) | **22.** (b) | **23.** (c) | **24.** (c) | **25.** (c) | **26.** (d) | **27.** (a) | **28.** (a) | **29.** (b) | **30**. (a) |
| **31.** (b) | **32.** (a) | **33.** (b) | **34.** (a) | **35.** (a) | **36.** (c) | **37.** (b) | **38.** (d) | **39.** (a) | **40**. (a) |
| **41.** (b) | **42.** (a) | **43.** (d) | **44.** (d) | **45.** (a) | **46.** (a) | **47.** (c) | **48.** (a) | **49.** (a) | **50**. (a) |
| **51.** (b) | | | | | | | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed August 2022, *https://www.learncpp.com*
- Codes Cracker, accessed August 2022, *https://codescracker.com*
- Roll Bar, accessed August 2022, *https://rollbar.com*
- Geeks For Geeks, accessed August 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed August 2022, *https://www.udacity.com*
- Scaler, accessed August 2022, *https://www.scaler.com*
- C Plus Plus, accessed August 2022, *https://cplusplus.com*
- Silly Codes, accessed August 2022, *https://sillycodes.com*

CHAPTER 11

# FILE HANDLING

## 11.1 Files and Streams

Organizing and maintaining our belongings helps us access them whenever they are needed with ease, and the same is true for our desktop folders and phone data. All need organizing and efficient handling. In C++, we handle and organize files, folders, and directories. We can write our own files and display the items in created files. What exactly is a file in C++? Let us try to define it in simple terms.

### Files

Files created in C++ help to store data onto the hard disk and remain in the system for later use. This is different from traditional programs, where our data or objects holding data get deleted after program termination to reclaim the temporary memory space allotted. Many times, we need our data to be securely stored in the system so fetching items out of a file requires less time.

### Streams

For the user to enter and provide details or values, we use input streams. A stream refers to a group of bytes that can be accessed sequentially. There are two different types of streams. *Input streams* are used to hold the input from a user, such as a keyboard. For example, the user might press any key on the keyboard, even when not asked for it. In a situation like this, all these keys are saved in the input stream and used later when the program itself requires it, saving the user energy and your code from any fault or error. The second type is an output stream, which is used for the output to a monitor, file, or printer. For example, let's suppose you need a printout, but the printer is currently working on another document, so your data/file waits for its turn to be given as output. To able to use all these functionalities in your C++ code, you need to include the `iostream` header file, which provides the program with a whole hierarchy of classes (multiple inheritance) to make use of all I/O classes.

Although the `iostream` class is derived from `ios_base`, iostream is the one you will be working directly with. The symbols << or >> are not to be confused with "greater than" or "less than." In C++, these are special operators you will add to your programs in your coding journey.

- The `iostream` class can handle both input and output streams, be it the user pressing random keys or an alert message to the user not to do so, all done through one class. It is the base class for all the other I/O stream classes we will be using throughout this chapter.
- The `istream` class is for input streams, and this class is derived from the `iostream` class. The *extraction operator* (>>) removes values from the stream created when the user presses keys (whether asked to or not). It also has functions for input operations like `get()` and `getline()`.
- The `ostream` class is for output streams and this class is derived from the `iostream` class. The *insertion operator* (<<) is used to put values in the stream to be displayed on output devices like monitors. It also has functions for input operations like `put()` and `write()`.
- The `fstream` class can handle both input and output operations related to files opening and closing, such as reading particular items of a file or writing onto a file. It has various functions with special operability.
- The `ifstream` handles all the file input-related operations and has various functions with special operability, like `get()`, `getline()`, `read()`, `seekg()`, and `tellg()`.
- The `ofstream` class can handle input operations related to writing onto a file. It has various functions with special operability like `put()`, `write()`, `seekp()`, and `tellp()`.
- The `streambuf` class is used to manage the input and output streams through a pointer, which points to the buffer.

### Code: Working with Files and Streams

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream ofObj;
    string ourLine;
    ofObj.open("textFile3.txt");
    while (ofObj)
    {
        getline(cin, ourLine);
        if (ourLine == "-1")
            break;
        cout<< ourLine << endl;
    }
    ofObj.close();
    ifstream ifObj;
    ifObj.open("textFile3.txt");
    while (ifObj)
    {
        getline(ifObj, ourLine);
        cout << ourLine << endl;
    }
    ifObj.close();
    return 0;
}
```

### Output:

```
We are learning and improving everyday.
We are learning and improving everyday.

-1
```

### Code Debriefing

- In this code, we used the `ofstream` class and created its object and a string value.
- We then used the `open` method to access our file in open mode.
- Text entered is written onto the file using the `ifstream` class object.

- The program then prints the content of the newly written file.
- The `close` method is called upon to close the file we had opened earlier.

## 11.2 File Operations

The major steps needed in C++ for file handling are as follows:

**Opening a file:** This is the first step taken toward file management in C++ and can be done either by passing our file name in the constructor when an object is created or using the `open()` method.

The syntax for opening a file is as follows:

```
void open(const char* ourFileName, ios::openMode mode);
or
open() function
```

Now, various open modes can be activated in C++.

*Table 11.1* Types of Open Modes

| Serial Number | Mode | What it does |
|---|---|---|
| **1** | `in` | This mode is the default while using `ifstream`, and it opens our file to read. |
| **2** | `out` | This mode is the default while using `ofstream`, and it opens our file to write. |
| **3** | `ate` | This mode opens our file, and the pointer faces the end of the file. |
| **4** | `binary` | This mode opens our file in binary form. |
| **5** | `noreplace` | This mode will open a file only if it does not exist in the system. |
| **6** | `nocreate` | This mode will open a file only if it exists in the system. |
| **7** | `trunc` | This mode will open a file if it already exists, and the items will be truncated. |
| **8** | `app` | This mode will open our file in append mode, and all new items will be appended at the end of the file. |

We can also combine different opening modes by separating each by using the symbol |, called the logical `or` symbol.

1. Writing to a file: Done using the `ofstream` or `fstream` classes to enter data onto our created or opened files.
2. Reading a file: Done using the `ifstream` or `fstream` classes to fetch data from our created or opened files.

3. Closing a file: This is the last, important step in file handling. At program termination, the memory is freed automatically. However, as to not take any chances and risk an item leak, we should always close our files.

### Code: Working with File Operations

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
fstream ourFile;
ourFile.open("OurFile",ios::out);
if(!ourFile)
{
cout<<"File was not created";
}
else
{
cout<<"Our new file has been created";
ourFile.close();
}
return 0;
}
```

**Output:**

```
Our new file has been created%
```

### Code Debriefing

- In this code, we used the `fstream` class and created its object.
- We used the `open` method to access our file in open mode.
- The program then prints the message "Our new file has been created."
- The `close` method is called upon to close the file we opened earlier.

### Code:Working with Text File

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
```

```
{
    ofstream ofObj;
    string ourLine;
    ofObj.open("textFile3.txt");
    while (ofObj)
    {
        getline(cin, ourLine);
        if (ourLine == "-1")
            break;
        cout<< ourLine << endl;
    }
    ofObj.close();
    ifstream ifObj;
    ifObj.open("textFile3.txt");
    while (ifObj)
    {
        getline(ifObj, ourLine);
        cout << ourLine << endl;
    }
    ifObj.close();
    return 0;
}
```

**Output:**

```
We are learning and improving everyday.
We are learning and improving everyday.

-1
```

### Code Debriefing

- In this code, we used the `ofstream` class and created its object and a string value.
- We then used the `open` method to access our file in open mode.
- Text entered is written onto the file using the `ifstream` class object.
- The program then prints the content of the newly written file.
- The `close` method is called upon to close the file we had opened earlier.

## 11.3 Random Access and Object Serialization

### Randomly Accessing a File

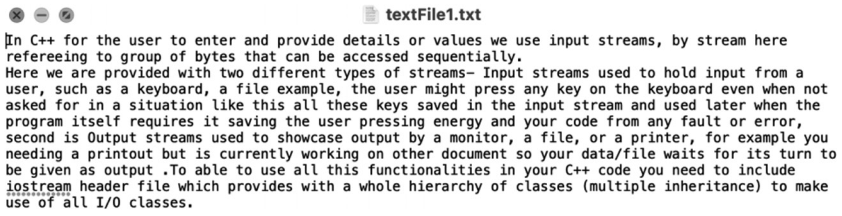
We have been accessing, opening, and fetching items from a file, either from the beginning or the end. However, in C++, we can also randomly access our opened or created files. *Random file access* means the control is not given by default to the start or the end of the file, but at any point in the file. This is done using special file operations that assign the file pointer to the point we wish to access.

- `Seekg()`: This function is used for input and `g` represents `get`; it will change the pointer's position to read.
- `Seekp()`: This function is used for output and `p` represents `post`; it will change the pointer's position to write.

Both of the above functions take up two parameter offsets that tell how far the pointer has to move in terms of bytes. The iostream flag tells where the new start is.

The syntax for randomly accessing a file is as follows:

```
fileObject seekg(offset number);
fileObject seekp(offset number);
```

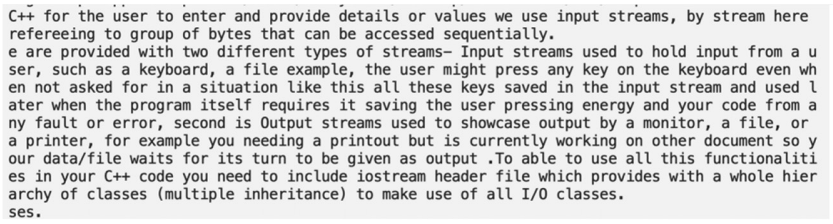


```
In C++ for the user to enter and provide details or values we use input streams, by stream here
refereeing to group of bytes that can be accessed sequentially.
Here we are provided with two different types of streams- Input streams used to hold input from a
user, such as a keyboard, a file example, the user might press any key on the keyboard even when not
asked for in a situation like this all these keys saved in the input stream and used later when the
program itself requires it saving the user pressing energy and your code from any fault or error,
second is Output streams used to showcase output by a monitor, a file, or a printer, for example you
needing a printout but is currently working on other document so your data/file waits for its turn to
be given as output .To able to use all this functionalities in your C++ code you need to include
iostream header file which provides with a whole hierarchy of classes (multiple inheritance) to make
use of all I/O classes.
```

### Code: Randomly Accessing a File

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    ifstream ifObj;
    ifObj.open( "textFile1.txt" );
    if (!ifObj)
    {
        cout << "Oops our file could not be opened\n";
        return 1;
    }

    string ourItem;
    ifObj.seekg(3);
    getline(ifObj, ourItem);
    cout << ourItem << '\n';
    ifObj.seekg(6, std::ios::cur);
    getline(ifObj, ourItem);
    cout <<ourItem << '\n';
    ifObj.seekg(-10, std::ios::end);
    getline(ifObj,ourItem);
    cout << ourItem << '\n';
    return 0;
}
```

**Output:**



```
C++ for the user to enter and provide details or values we use input streams, by stream here
refereeing to group of bytes that can be accessed sequentially.
e are provided with two different types of streams- Input streams used to hold input from a u
ser, such as a keyboard, a file example, the user might press any key on the keyboard even wh
en not asked for in a situation like this all these keys saved in the input stream and used l
ater when the program itself requires it saving the user pressing energy and your code from a
ny fault or error, second is Output streams used to showcase output by a monitor, a file, or
a printer, for example you needing a printout but is currently working on other document so y
our data/file waits for its turn to be given as output .To able to use all this functionaliti
es in your C++ code you need to include iostream header file which provides with a whole hier
archy of classes (multiple inheritance) to make use of all I/O classes.
ses.
```

### Code Debriefing

- In this code, we used the `ifstream` class and created its object and a string value.
- We then used the `open` method to access our file in open mode.
- The `Seekg` method is used to print the content of the newly written file.
- The `getline` method helps to print the items of our file line by line, and it takes two parameters.

### Object Serialization

*Serialization* in C++ programming is the mechanism of converting an object in our code into a sequence of bytes (also called the stream of data or the object). This stream can be stored easily and even used for communicating between systems and sharing information contained in the sequence of bytes. *Deserialization* is done to reverse the process of serialization, where the bytes are converted back to the objects and normal data.

## Summary

- Files created in our C++ program help to store data onto the hard disk and remain in the system for later use.
- In C++, for the user to enter and provide details or values, we use input streams. A *stream* references a group of bytes that can be accessed sequentially.
- *Input streams* are used to hold the input from a user, such as a keyboard.
- *Output streams* are used to showcase output by a monitor, file, or printer.
- In C++ code, you need to include the `iostream` header file, which provides the program with a whole hierarchy of classes (multiple inheritance) to make use of all I/O classes.
- The `iostream` class can handle both input and output streams, even if it is a user pressing random keys.
- The `fstream` class can handle both input and output operations related to files opening and closing, like reading.
- Opening a file is the first step taken toward file management in C++. It can be done either by passing a file name in the constructor when an object is created or using the `open()` method.
- Writing to a file is done using the `ofstream` or `fstream` classes to enter data onto created or opened files.
- Reading a file is done using the `ifstream` or `fstream` classes to fetch data from created or opened files.
- Closing a file is the last, important step in file handling.
- At program termination, memory is freed automatically. However, as to not take any chances to have an item leak, we should always close our files.
- *Random file access* means the control is not given by default to the start or the end of the file, but at any point in the file. This is done using special file operations that assign the file pointer to the point we wish to access.

- The `Seekg()` function is used for input, and `g` represents `get`. It will change the pointer position to read.
- The `Seekp()` function is used for output, and `p` represents `post`. It will change the pointer position to write.
- *Serialization* is the mechanism of converting an object into a sequence of bytes (also called a stream of data).
- *Deserialization* is also done to reverse the process of serialization, where the bytes are converted back to the objects and normal data.</UL>

## Exercises

### Theory Questions

1. What is file handling in C++? Discuss its various aspects.
2. What streams are required while handling files?
3. Discuss seek functions and its form in C++.
4. How is random access carried out in a file?
5. Explain the various modes of opening writing and reading a file?
6. What is serialization in C++? How is it carried out?
7. What is deserialization in C++? How is it carried out?

### Practical Questions

1. Write a C++ program to maintain a book record using file handling.
2. Write a C++ program to maintain house records using file handling.
3. Write a C++ program for the registration (signup) process using file handling.
4. Write a C++ program to read and write file operations in file handling.
5. Write a C++ menu-driven program to perform the following actions: add, modify, append, and display.
6. Write a C++ program to store or enter data to a file using file handling.
7. Write a C++ program to retrieve information from the file using file handling.

8. Write a C++ program to read and display a file using file handling.
9. Write a C++ program to merge two files into a third file using file handling.
10. Write a C++ program to encrypt files using file handling.
11. Write a C++ program to decrypt files using file handling.
12. Write a C++ program to read and write values through an object using file handling.
13. Write a C++ program to count digits, letters, and spaces using file handling.
14. Write a C++ program to count words, lines, and total size using file handling.
15. Write a C++ program to read a text file and write it in another text file using file handling.
16. Write a C++ program to count the occurrence of a word using file handling.
17. Write a C++ program to read and write student details using file handling.
18. Write a C++ program to manipulate file pointers using file handling.

**MCQ-Based**

1. By default, all the files are opened in _________ mode.

   **a.** Binary

   **b.** Text

   **c.** Cannot be determined

2. It is not possible to combine two or more file opening modes in the open () method.

   **a.** True

   **b.** False

3. Which of the following is not a file opening mode?

   **a.** `ios::ate`

**b.** `ios::nocreate`

**c.** `ios::noreplace`

**d.** `ios::truncate`

**4.** Due to ios::trunc mode, the file is truncated to zero length.

**a.** True

**b.** False

**5.** If we have an object from the ofstream class, then the default mode of opening the file is _________.

**a.** `ios::in`

**b.** `ios::out`

**c.** `ios::in|ios::trunc`

**d.** `ios::out|ios::trunk`

**6.** _________ is the return type of the open() function.

**a.** `int`

**b.** `bool`

**c.** `float`

**d.** `char*`

**7.** If we have an object from the fstream class, then what is the default mode of opening the file?

**a.** `ios::in|ios::out`

**b.** `ios::in|ios::out|ios::trunc`

**c.** `ios::in|ios::trunc`

**d.** Default mode depends on the compiler

**8.** To create an output stream, we must declare the stream to be of class _________.

**a.** `ofstream`

**b.** `ifstream`

**c.** `iostream`

**d.** None of these

**9.** Streams that will be performing both input and output operations must be declared as class _________.

**a.** `iostream`

**b.** `fstream`

**c.** `stdstream`

**d.** `Stdiostream`

**10.** To perform file I/O operations, we must use the _________ header file.

**a.** `< ifstream>`

**b.** `< ofstream>`

**c.** `< fstream>`

**d.** Any of these

**11.** Which of the following is not used to seek a file pointer?

**a.** `ios::cur`

**b.** `ios::set`

**c.** `ios::end`

**d.** `ios::beg`

**12.** Which of the following is used to move the file pointer to the start of a file?

**a.** `ios::cur`

**b.** `ios::start`

**c.** `ios::first`

**d.** `ios::beg`

**13.** Which of the following is not used as a file opening mode?

**a.** `ios::trunc`

**b.** `ios::binary`

   **c.** `ios::in`

   **d.** `ios::ate`

**14.** Which stream class only writes on files?

   **a.** `ofstream`

   **b.** `ifstream`

   **c.** `fstream`

   **d.** `iostream`

**15.** Which of these is the correct statement about `eof()`?

   **a.** Returns true if a file open for reading has reached the next character.

   **b.** Returns true if a file open for reading has reached the next word.

   **c.** Returns true if a file open for reading has reached the end.

   **d.** Returns true if a file open for reading has reached the middle.

**16.** Which of the following true about `FILE *fp`?

   **a.** `FILE` is a structure, and fp is a pointer to the structure of the `FILE` type.

   **b.** `FILE` is a buffered stream.

   **c.** `FILE` is a keyword in C++ for representing files, and `fp` is a variable of the `FILE` type.

   **d.** `FILE` is a stream.

**17.** Which of the following methods can be used to open a file in file handling?

   **a.** Using `open( )`

   **b.** Constructor method

   **c.** Destructor method

   **d.** Both A and B

**18.** Which operator is used to insert data into a file?

   **a.** >>

   **b.** <<

**c.** <

**d.** None of the above

**19.** Which is the correct syntax?

**a.** `myfile:open ("example.bin", ios::out);`

**b.** `myfile.open ("example.bin", ios::out);`

**c.** `myfile::open ("example.bin", ios::out);`

**d.** `myfile.open ("example.bin", ios:out);`

**20.** What is the output of this program?

**NOTE** *Includes all required header files*

```
 using namespace std;
  int main ()
{
    int l;
    char * b;
    ifstream i;
    i.open ("find.txt", ios :: binary );
    i.seekg (0, ios :: end);
    l = i.tellg();
    i.seekg (0, ios :: beg);
    b = new char [l];
    i.read (b, l);
    i.close();
    cout.write (b, l);
    delete[] b;
    return 0;
 }
```

**a.** Error

**b.** find

**c.** This is find

**d.** Runtime error

**21.** What is the output of this program?

**NOTE** *Includes all required header files*

```
using namespace std;
int main ()
 {
    char fine, course;
    cout << "Enter a word: ";
   fine = cin.get();
 cin.sync();
 course = cin.get();
 cout << fine << endl;
 cout << course << endl;
 return 0;
 }
```

**a.** course

**b.** fine

**c.** Returns "fine" and Two letters or numbers from the entered word

**d.** None of the mentioned

**22.** What is `ios :: trunc` used for ?

**a.** If the file is opened for output operations and it already existed, no action is taken.

**b.** If the file is opened for output operations and it already existed, then a new copy is created.

**c.** If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

**d.** None of the above

**23.** What will be the output of the following program?

**NOTE** *Includes all required header files*

```
 using namespace std;
 int main()
{
   ofstream ofile;
   ofile.open ("find.txt");
```

```
    ofile << "letsfindcourse" << endl;
    cout << "Data written to file" << endl;
    ofile.close();
     return 0;
  }
```

**a.** Compile error

**b.** Runtime error

**c.** The program prints “”letsfindcourse”” in the file find.txt

**d.** None of the above

**24.** What is the output of this program?

**NOTE** *Includes all required header files*

```
using namespace std;
int main ()
{
  char fine, course;
  cout << "Enter a word: ";
  fine = cin.get();
 cin.sync();
 course = cin.get();
 cout << fine << endl;
 cout << course << endl;
 return 0;
}
```

**a.** course

**b.** fine

**c.** Returns fine and 2 letters or numbers from the entered word

**d.** None of the above

**25.** What is the output of this program?

**NOTE** *Includes all required header files*

```
using namespace std;
int main ()
 {
   ofstream outfile ("find.txt");
   for (int i = 0; i < 70; i++) {
           outfile << i; outfile.flush();
     }
cout << "Done";
outfile.close();
 return 0;
 }
```

**a.** Done

**b.** Error

**c.** Runtime error

**d.** None of the above

**26.** What is the output of this program?

**NOTE** *Includes all required header files*

```
using namespace std;
int main ()
{
   int p = 1000;
   double q = 3.14;
   cout << p;
   cout << endl;
   cout << q << endl << p * q;
   endl (cout);
   return 0;
 }
```

**a.** 1000

**b.** 3.14

**c.** 3140

**d.** All of the above

**27.** Which of the following is true about the following program

**NOTE** *Includes all required header files*

```
 using namespace std;
 int main ()
 {
   char i;
   streambuf * p;
   ofstream of ("find.txt");
   pbuf = of.rdbuf();
   do { i = cin.get();
   p -> sputc(i);
   } while (i != '.');
   of.close();
   return 0; }
```

**a.** insertion operator

**b.** $ symbol

**c.** dot operator

**d.** none of the above

**28.** What will be the output of this program?

**NOTE** *Includes all required header files*

```
 using namespace std;
 int main () {
 int p = 10;
 double q = 1.14;
 cout << p + q;
 endl (cout);
return 0;
}
```

**a.** 10

**b.** 1.14

**c.** 11.14

**d.** All of the above

**29.** What will be the output of this program?

**NOTE** *Includes all required header files*

```
 using namespace std;
 int main () {
FILE *fp;
char x[1024];
fp = fopen("find.txt", "r");
// "Mary and Brit" x[0] = getc(fp);
fseek(fp, 0, SEEK_END);
fseek(fp, -7L, SEEK_CUR);
fgets(x, 6, fp);
puts(x);
return 0;
}
```

**a.** Mary

**b.** Harry

**c.** Chris

**d.** Brit

**30.** In `fopen()`, the open mode "wx" is sometimes preferred over "w" because

**i.** Use of wx is more efficient.
**ii.** If w is used, the old contents of the file are erased and a new empty file is created. When wx is used, `fopen()` returns NULL if the file already exists.

**a.** Only i

**b.** Only ii

**c.** Both i and ii

**d.** None of the above

**31.** Which member function is used to determine whether the stream object is currently associated with a file?

**a.** `is_open`

**b.** `Buf`

**c.** `String`

**d.** None of the above

**32.** `getc()` returns `EOF` when

**a.** End of files is reached

**b.** When `getc()` fails to read a character

**c.** Both A and B

**d.** None of the above

**33.** `fseek()` should be preferred over `rewind()` mainly because

**a.** In `rewind()`, there is no way to check if the operations were completed successfully

**b.** `rewind()` does not work for empty files

**c.** `rewind()` does work for empty files

**d.** All of the above

**34.** Which function is used to return to the first position back from the end of a file object?

**a.** `Seekg`

**b.** `Seekp`

**c.** Both seekg and seekp

**d.** None of the above

**35.** Which among following is used to open a file in binary mode?

**a.** `ios:app`

**b.** `ios::out`

**c.** `ios::in`

**d.** `ios::binary`

**36.** How can we get to the position at the nth byte of fileObject?

**a.** `fileObject.seekg( 'filename',n );`

**b.** `fileObject.seekg( n, 'filename' );`

**c.** `fileObject.seekg( n );`

**d.** `fileObject.seekg( n, ios::app );`

**37.** Where is a file temporarily stored before a read or write operation is performed in C++?

**a.** RAM

**b.** Notepad

**c.** Buffer

**d.** Hard disk

## MCQ

| Answer Key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1**. (b) | **2.** (b) | **3.** (d) | **4.** (a) | **5.** (d) | **6.** (b) | **7.** (d) | **8.** (a) | **9.** (b) | **10.** (c) |
| **11**. (b) | **12.** (d) | **13.** (a) | **14.** (a) | **15.** (c) | **16.** (a) | **17.** (d) | **18.** (b) | **19.** (b) | **20.** (d) |
| **21**. (c) | **22.** (c) | **23.** (c) | **24.** (a) | **25.** (b) | **26.** (d) | **27.** (d) | **28.** (d) | **29.** (d) | **30.** (b) |
| **31**. (a) | **32.** (c) | **33.** (a) | **34.** (a) | **35.** (d) | **36.** (c) | **37.** (c) | | | |

## References

### Books

- B. Stroustrup, *The C++ Programming Language* (4th Edition) (Addison-Wesley Professional, 2013).
- K. R. Venugopal, *Mastering C++* (2nd Edition) (McGraw Hill Education, July 2017).
- Y. Kanetkar, *Let Us C++* (BPB Publications September, 2020).
- Y. Kanetkar, *Test Your C++ Skills* (BPB Publications March, 2003).

### Websites

- Learn CPP, accessed August 2022, *https://www.learncpp.com*
- Codes Cracker, accessed August 2022, *https://codescracker.com*
- Geeks For Geeks, accessed August 2022, *https://www.geeksforgeeks.org*
- Udacity, accessed August 2022, *https://www.udacity.com*
- Scaler, accessed August 2022, *https://www.scaler.com*
- C Plus Plus, accessed August 2022, *https://cplusplus.com*
- Silly Codes, accessed August 2022, *https://sillycodes.com*

# INDEX