University of Mumbai

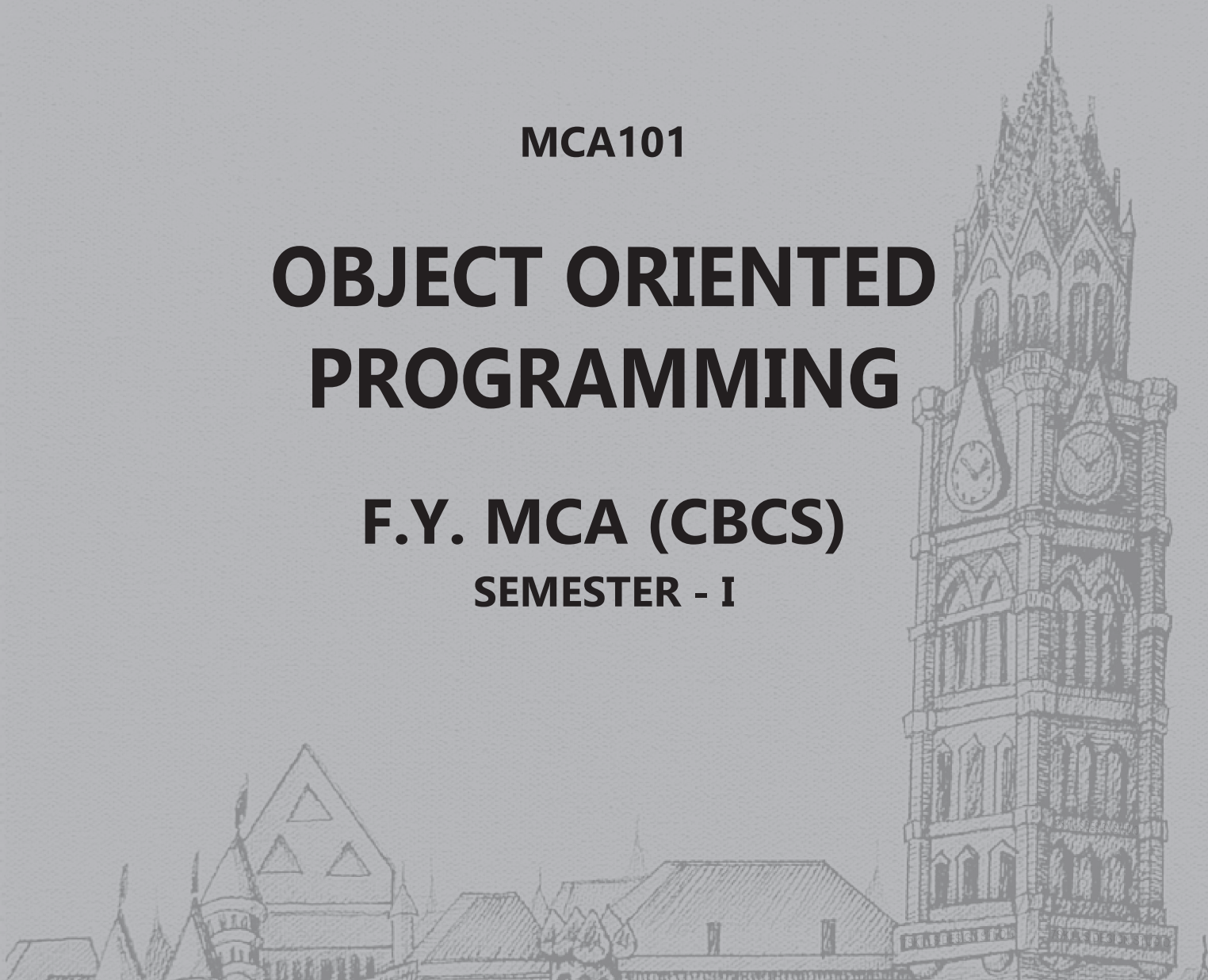
INSTITUTE OF DISTANCE AND OPEN LEARNING

MCA101

OBJECT ORIENTED PROGRAMMING

F.Y. MCA (CBCS)

SEMESTER - I



Object Oriented Programming

F.Y. MCA (CBCS)
Semester - I

Prof. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai

Prof. Prakash Mahanwar
Director Incharge
Institute of Distance & Open Learning,
University of Mumbai, Mumbai

Prof. Madhura Kulkarni
Asst. Director & Incharge
Study Material Section,
University of Mumbai, Mumbai

Programme Co-ordinator, I/c Faculty of Science & Technology

Prof. Mandar Bhanushe
Department of Mathematics, University of Mumbai, IDOL

Course Co-ordinator

Asst. Prof. Reshma Kurkute
Department - MCA, University of Mumbai, IDOL

Asst. Prof. Shardul Gavande
Department - MCA, University of Mumbai, IDOL

Course Writer

Asst. Prof. Nikhil Pawnikar
Department - UDIT, University of Mumbai, IDOL

Asst. Prof. Dhanraj Jadhav
Department - UDIT, University of Mumbai, IDOL

Asst. Prof. Seema Vishwakarma
Vidyalankar School of Information Technology, Wadala East, Mumbai.

Asst. Prof. Madhavi Auoralkar
Vidyalankar School of Information Technology, Wadala East, Mumbai.

Asst. Prof. Milind Thorat
K. J. Somaiya Institute of Engineering & I.T., Sion. Mumbai.

Published by
Dr. Prakash Mahanwar, Director Incharge
on behalf of Institute of Distance & Open Learning,University of Mumbai, Mumbai
and Printed at Printers Name
Printers address

DTP Composed by M/s 7SKILLS

CONTENTS

Object Oriented Programming

F.Y. MCA (CBCS)
Semester - I

SYLLABUS

Sr. No.	Module	Detailed Contents	Hours
1	**Programming Basics**	Introduction to Programming, Programming Paradigms, Programming Languages and Types. Introduction to C - Basic Program Structure, Execution flow of C Program, Directives, Basic Input /Output. Introduction to Object Oriented Programming- OOP concepts, Advantages, Applications, Comparison of C and C++-Data Types, Control Structures, Operators and Expressions	**8**
2	**Introduction to C++**	Structure of a C++ program, Execution flow, Classes and Objects, Access modifiers, Data Members, Member Functions, Inline Functions, Passing parameters to a Function(pass by Value, Pass by Address, Pass by Reference), Function with default arguments, Function Overloading, Object as a Parameter, Returning Object Static data members and functions, Constant Data members and functions Constructors - Default, Parameterized, Copy, Constructor Overloading, Destructors Arrays, Array as a Class Member, Array of Objects, Strings-Cstyle strings and String Class	**10**
3	**Operator Overloading and Pointers**	Operator Functions-Member and Non Member Functions, Friend Functions Overloading Unary operators Overloading binary operators(Arithmetic, Relational, Arithmetic Assignment, equality), Overloading Subscript operator. Type Conversion Operators- primitive to Object, Object to primitive, Object to Object.	**10**

Sr. No.	Module	Detailed Contents	Hours
		Disadvantages of operator Overloading, Explicit and Mutable Pointers, Pointer and Address of Operator, Pointer to an Array and Array of Pointers, Pointer arithmetic, Pointer to a Constant and Constant Pointer, Pointer Initialization, Types of Pointers(void, null and dangling), Dynamic Memory. Allocation, Advantages and Applications of pointers	
4	**Inheritance and Polymorphism**	Inheritance Concept, Protected modifier, Derivation of Inheritance- Public, Private and Protected, Types of Inheritance-Simple, Multilevel, Hierarchical, Multiple, Hybrid, Constructors and Inheritance, Function Overriding and Member hiding. Multiple Inheritance, Multipath inheritance – Ambiguities and solutions. Polymorphism, Static and Dynamic Binding, Virtual Functions, Pure Virtual Functions, Virtual destructors, Abstract Classes, Interfaces.	**8**
5	**Streams and**	Files, Text and Binary Files, Stream Classes, File IO using.	**8**
	Exceptions	Stream classes, File pointers, Error Streams, Random File Access, Manipulators, Overloading Insertion and extraction operators Error handling, Exceptions, Throwing and catching exceptions, Custom Exceptions, Built in exceptions	
6	**Advanced C++**	Casting- Static casts, Const Casts, Dynamic Casts, and Reinterpret Casts. Creating Libraries and header files. Namespaces Generic Programming, Templates, Class Templates, Function Templates, Template arguments, STL Database Programming with MySQL	**8**

UNIT 1

1

PROGRAMMING BASICS

Unit Structure

1.1 Objectives

- To understand how C++ improves C with object-oriented features.
- To learn different types of language available.
- To understand the structure of C++ Programming Language.
- To learn the concept of C++ programming language.
- To learn the syntax and semantics of the C++ programming language.
- To understand the difference between C & C++.

1.2 Introduction to Programming

- The program is a sequence or step by step of instruction along with data.
- A program is constituted by two fundamental parts:
- A representation of the information relative to the domain of interest: object
- Description of how to manipulate the representation in such a way as to understand the desired functionality: Operations.
- The programming language has two components
    1. syntax and
    2. semantics

1.3 Programming Paradigms

- Paradigm is a method to solve some problem or do some task.
- A programming paradigm is an approach to solve the problem using some programming language tools and techniques
- **Types of programming Paradigm**

1. Imperative programming paradigm: Features of Imperative programming is a close relation to machine architecture. It is based on Von Neumann architecture. Through assignment statements, you can change the program state. Using this paradigm-changing the state step by step. The main focus of

this paradigm is on how to achieve the goal? The paradigm consists of several statements and after execution, the result is stored.

Advantage:

1. Very simple to implement
2. It contains loops, variables, etc.

Disadvantage:

1. A complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

An imperative programming paradigm is divided into three categories: Procedural, OOP and parallel processing

a. Procedural programming paradigm –This paradigm focused on the procedure in terms of the underlying machine model. Both procedural and imperative approaches are similar. It can reuse the code.

b. Object-oriented programming –The OOPs are a collection of classes and objects which communicate with each other. The object is the smallest and basic entity and all kinds of computation are performed on the objects only. Emphasis is on data rather than procedure. It can handle almost all kinds of real-life problems that are today in the scenario.

Advantages:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

c. Parallel processing approach – In this approach, the program instructions are divided into multiple processors. A parallel processing system has many numbers of processors to run a program in less time by dividing them. This approach follows a divide and conquers method.

2. Declarative programming paradigm: It is divided into Logic, Functional & Database. *Declarative programming* is a style of building programs that express the logic of computation never talking about its control flow. It is simpler by writing parallel programs. The focus is on data (what needs to be done) rather than the procedure (how it should be done). It just declares the result we want rather than how it has been produced. This is the only difference between imperative and declarative programming paradigms.

a. Logic programming paradigms

Logic programming is an abstract model of computation. Using this programming paradigm you can solve logical problems like puzzles, series, etc. Logic programming is a knowledge base that we know before and along with the question and knowledge base which is given to machines, it produces results. In logical programming, the main significance is on the knowledge base and the problem. The execution of the program is extremely similar to the proof of the mathematical statement, e.g., Prolog

b. Functional programming paradigms

Functional programming is language independent. The main aim of this paradigm is the execution of a series of mathematical functions. In this paradigm, data is loosely coupled to functions. The function hides its implementation. The function can be replaced with their values or parameters without changing the meaning of the program. Some of the languages like Perl, JavaScript mostly use this paradigm.

c. Database/Data-driven programming approach

This programming technique is based on data and its movement. Program statements are defined by data rather than hard-coding a series of instructions. Using a database program you can create files, make data entry, update, query, and reporting functions. Several programming languages are developed mostly for database applications. For example SQL.

1.4 Programming Languages and Types

A programming language is a formal language composed of a set of instructions. that produce different output depending on the user requirements. Programming languages are used for implementing algorithms. programming languages consist of a set of instructions.

Types of computer programming languages, they are

1. Low-level programming languages
2. High-level programming languages
3. Middle-level programming languages

1) Low-level programming languages

- Low-level programming are machine-dependent programming languages such as Binary (Machine code) and Assembly language.
- Computers only understand the Binary language for example 0's and 1's (High or Low), so these languages are the best way to give signals to the computer directly.
- Machine Code does not need any interpreter or compiler to convert language in any form because the computer understands these signals directly.
- For computers to understand the instructions written in Assembly language that needs to be converted from Assembly language into equivalent Binary.
- Low-Level programming language programs are faster than High-Level programming language programs as they have fewer keywords, symbols, and no need to convert into other Code.

2) High-level programming languages

- High-level programming languages are machine-independent programming languages, which are easy to write, read, edit, and understand.
- Languages like Java, .Net, Pascal, COBOL, C++, C, C# These languages come under the high-level programming language category.
- These programming languages have some special keywords, functions, and class libraries. We can easily build a program for the computer using this language.

Features of High-Level programming languages

- platform Independent
- Easy to understand
- Easy to code, read and edit
- Popular to develop User End Applications.

3) Middle-Level programming language

- Middle-Level Programming combines the features of low level and high-level programming languages.
- These programming languages have features of Low Level as well as High-Level programming languages known as "Middle Level" programming languages.
- E.g. C programming languages

1.5 Introduction to C

1.5.1 Basic Program Structure

A c program consists of the following sections −

- Pre-processor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Example:

Header file	#include<stdio.h>
Main() function	int main() {
Variable declaration	int a=10;
Body	printf("%d", a);
Return	return 0; }

1.5.2 Execution flow of C Program

1. source code is sent to the preprocessor. The preprocessor is convert preprocessor directives into their respective values. The preprocessor generates source code.

2. Expanded source code is sent to the compiler which compiles the code and converts it into assembly code.
3. The assembly code is sent to the assembler which assembles the code and converts it into object code.
4. The object code is converted into executable code with the help of linker which links it to the library such as header files.
5. The executable code is sent to the loader which loads it into memory and then it is executed. After execution, output is sent to the console.

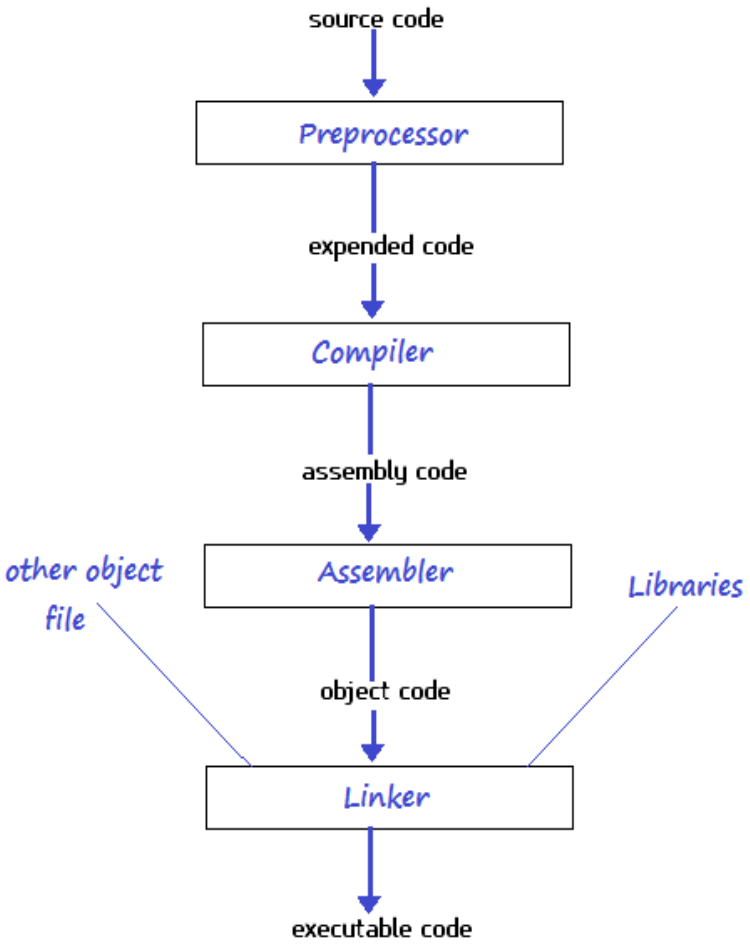


fig.1 Execution flow of C Program

1.5.3 Directives

directive specifies how a compiler should process its input.

Table 1: List of Preprocessor directives

Directive	Function
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the file to be include
#ifdef	Tests for macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile time condition
#else	specifies alternatives when #if fails

1.5.4 Basics Input / Output

a. In C, printf() is one of the main output functions. The function sends formatted output to the screen. For example,

Example

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      printf("C Programming");
5.      return 0;
6.  }
```

b. In C, scanf() is one of the commonly used functions to take input from the user. The scanf() function reads formatted input from the standard input such as keyboards.

Example:

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int testInt;
5.      printf("Enter an integer: ");
6.      scanf("%d", &testInt);
7.      printf("Number = %d",testInt);
8.      return 0;
9.  }
```

1.6 Introduction to Object-Oriented Programming

1.6.1 OOP concepts

In OOPs, the Decomposition of a problem into several smaller units (entities) called objects and then builds data and functions around these objects. Before Object-Oriented Programming came, programs were written in a procedural language, they were nothing but a long list of instructions. On the other hand, these objects can interact with each other; this makes it easier to develop programs in OOP as we can understand the relationship between them.

The organization of data and function in Object-Oriented Programming shown in fig.

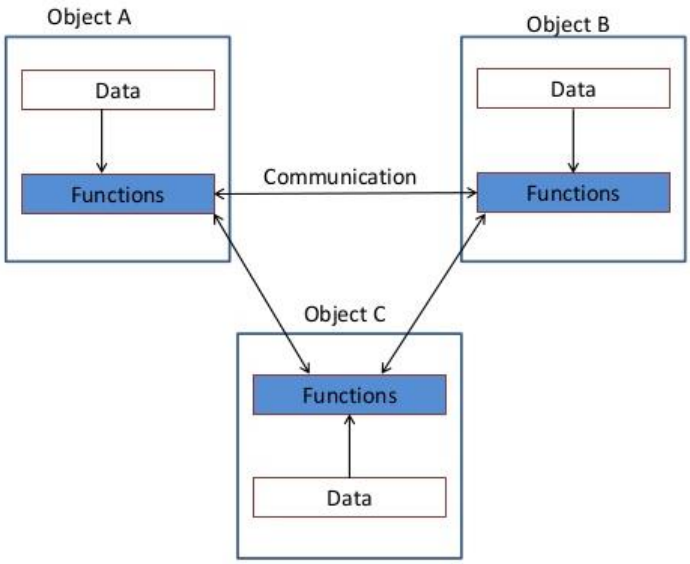


Fig.2 organization of data and function

Features of Object-Oriented Programming:

a. Importance of data rather than the procedure

b. The program is divided into small units known as objects.

c. Data is hidden

d. Object communicate with each other through objects

e. Follows bottom-up approach

f. New data and function can easily be added

Class and Objects

A class is like a blueprint of a data member and functions and objects are an instance of the class. For example, let's say we have a class **Car** that has data members (variables) such as speed, weight, price, and functions such as gearChange(), slowDown(), brake(), etc. Now let's say I create an object of this class named FordCar which uses these data members and functions and gives them its values. Similarly, we can create as many objects.

Example:

```
class Car
{
   //Data members
   char name[20];
   int speed;
   int weight;

public:
   //Functions
   void brake(){
   }
   void slowDown(){
   }
};
```

```
int main()
{
  //ford is an object
  Car ford;
}
```

Abstraction

Abstraction is a process of hiding background details from the user. For example, When you send an SMS you just type the message, select the contact and click send, the phone shows you that the message has been sent, what happens in the background, when you click send button is hidden from you as it is not relevant to you. Since classes use the concept of data abstraction, are known as Abstract Data Type (ADT).

Encapsulation

The process of combining data and function into a single unit is known as Encapsulation. This will not allow access to private data members from outside the class. To achieve encapsulation, we make all data members of class private and create public functions, using them we can get the values from these data members or set the value to these data members.

Inheritance

Objects of one class obtain the properties of objects of another class. in other words, accessing the property of parents (base) class from child class (derived).

Example:

```
#include <iostream>
using namespace std;
class ParentClass {
  //data member
  public:
    int varone =100;
```

```
};
class ChildClass: public ParentClass {
 public:
 int vartwo = 500;
};
int main(void) {
 ChildClass obj;
}
```

Polymorphism

Examples of polymorphism are Function overloading and Operator overloading. Polymorphism is a feature using which an object behaves differently in different situations. In function overloading, we can have more than one function with the same name but different numbers, type or sequence of arguments.

Example:

```
#include <iostream>
using namespace std;
class Sum {
 public:
   int add(int num1,int num2){
     return num1 + num2;
   }
   int add(int num1, int num2, int num3){
     return num1 + num2 + num3;
   }
};
int main(void) {
  //Object of class Sum
  Sum obj;
```

```
  //This will call the second add function
  cout<<obj.add(10, 20, 30)<<endl;
  //This will call the first add function
  cout<<obj.add(11, 22);
  return 0;
}
```

Output:

```
60
33
```

1.6.2 Advantages

a. **Simplicity:** the complexity is reduced and the simple program structure.

b. **Modularity:** each object forms a separate entity in oops.

c. **Modifiability:** Easy to make minor changes in the data representation. modification inside a class do not affect any other part of a program since the only public interface that the external world has to a class is through the use of methods

d. **Extensibility:** adding new features or introducing a few new objects and modifying some existing ones

e. **Maintainability:** objects can be maintained separately, making locating and fixing of problems become easier

f. **Re-usability:** objects can be reused in different programs

1.6.3 Applications

1. Client-Server Systems

Object-oriented Client-Server Systems provide the operating systems, networks, and hardware, creating object-oriented Client-Server Internet (OCSI) applications.

2. OSCI consist of three major technologies:

- The Client Server
- Object-Oriented Programming
- The Internet

3. Object-Oriented Databases

Object Database Management Systems databases store objects instead of data, such as real numbers and integers. Objects consist of the following:

Attributes: Attributes are data that define the traits of an object. This data can be as simple as integers and real numbers. It can also be an allusion to a complex object.

Methods: methods define the behavior and are also called functions or procedures.

4. Real-Time System Design

Real-time systems have inherent complexities that make it difficult to build them. These Object-Oriented techniques make it easier to handle those complexities. These techniques use an integrated framework for presenting ways of dealing with these complexities by providing an that includes schedulability analysis and behavioral specifications.

5. Simulation And Modelling System

Due to the varying specification of variables, It's difficult to model complex systems These are prevalent in medicine and other areas of natural science, such as ecology, zoology, and agronomic systems. Simulating complex systems requires modeling and understanding interactions. Object-oriented Programming provides an alternative approach for simplifying these complexity systems.

6. Hypertext And Hypermedia

In OOPs, Hypertext is similar to the regular text as it can be stored, searched, and edited easily. The only difference is that hypertext is text with pointers to other text as well.

Hypermedia, on the other hand, is a superset of hypertext. Documents having hypermedia, not only contain links to other pieces of text and information, but also numerous other forms of media, ranging from images to sound.

7. Neural Networking And Parallel Programming

It addresses the problem of prediction and approximation of complex time-varying systems in OOPs. At start the entire time-varying process is split into several time intervals or slots. Then, neural networks are developed in a particular time interval to disseminate the load of various networks. OOP simplifies the whole process by simplifying the approximation and prediction ability of networks.

8. Office Automation Systems

This system includes formal as well as informal electronic systems, concerned with information sharing and communication to and from people inside as well as outside the organization. Some examples are:

- Email
- Word processing
- Web calendars
- Desktop publishing

9. CIM/CAD/CAM Systems

OOP can also be used in manufacturing and design applications as it allows people to reduce the try involved. For occasion, it can be used while designing blueprints, flowcharts, etc. It is possible for the designers and engineers to produce these flowcharts and blueprints accurately with help of OOPs..

10. AI Expert Systems

These are computer applications which are developed to solve complex problems about a specific domain, which is at a level far beyond the reach of a human brain.

1.6.4 Comparison of C and C++

Table.2 Difference between C & C++

C	C++
C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.
C does not support polymorphism, encapsulation, and inheritance which means that C does not support object-oriented programming.	C++ supports polymorphism, encapsulation, and inheritance because it is an object-oriented programming language.
C is a subset of C++.	C++ is a superset of C.
C contains 32 keywords.	C++ contains 52 keywords.

For the development of code, C supports procedural programming.	C++ is known as a hybrid language because C++ supports both procedural and object-oriented programming paradigms.
Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in the form of an object in C++.
C does not support information hiding.	Data is hidden by the Encapsulation to ensure that data structures and operators are used as intended.
Built-in data types are supported in C.	A built-in & user-defined data type is supported in C++.
C is a function-driven language because C is a procedural programming language.	C++ is an object driven language because it is object-oriented programming.
Function and operator overloading are not supported in C.	Function and operator overloading are supported by C++.
C is a function-driven language.	C++ is an object-driven language
Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Namespace features are not present inside the C.	A namespace is used by C++, which avoids name collisions.
The header file used by C is stdio.h.	The header file used by C++ is iostream.h.
Reference variables are not supported by C.	Reference variables are supported by C++.
Virtual and friend functions are not supported by C.	Virtual and friend functions are supported by C++.
C does not support inheritance.	C++ supports inheritance.
Instead of focusing on data, C focuses on method or process.	C++ focuses on data instead of focusing on method or procedure.

C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory deallocation.	C++ provides a new operator for memory allocation and delete operator for memory deallocation.
Direct support for exception handling is not supported by C.	Exception handling is supported by C++.
scanf() and printf() functions are used for input/output in C.	cin and cout are used for input/output in C++.

1.6.5 Data Types

Data types in C++ is mainly divided into three parts

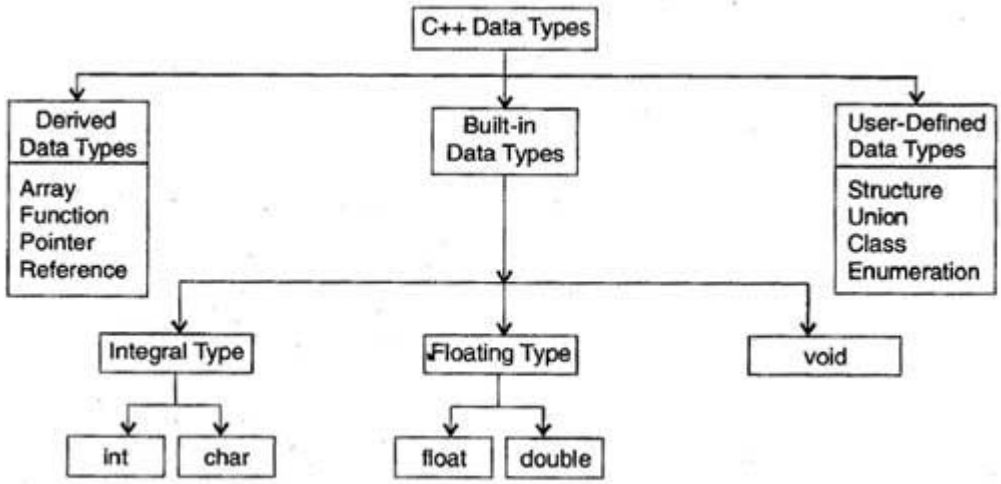


Fig.3 Various data types in c++

1. **Primitive Data Types**: These data types are built-in types or predefined data types and can be used by the user to declare variables.

Example: int, char, float, bool, etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

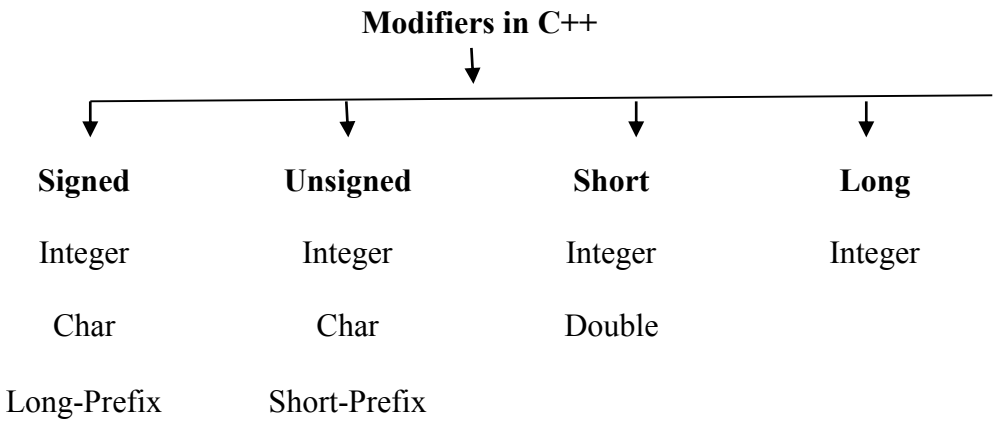
2. **Derived Data Types:** These data-types derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
    - Function
    - Array
    - Pointer
    - Reference
3. **Abstract or User-Defined Data Types**: These data types are defined by the user itself. which is the same as defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
    - Class
    - Structure
    - Union
    - Enumeration
    - Typedef defined DataType

- **Integer**: The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character**: This data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean**: This data type is used for storing boolean values. A boolean variable can store value either *true* or *false*. The keyword used for the boolean data type is **bool**.
- **Floating Point**: This data type holds a real number and used for storing single-precision floating-point values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
- **Double Floating Point**: this data type is used for storing decimal values. The keyword used for the double floating-point data type is **double**. Double variables require 8 bytes of memory space.

- **void**: Void means without any value. void data type represents a valueless entity. The void data type is used for those functions which do not return a value.

Data Type Modifiers

As the name implies, data type modifiers are used with built-in data types to modify the length of data that a particular data type can hold.



Below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Table 3: Modifiers in C++

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	

Data Type	Size (in bytes)	Range
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Note: Above values may vary from compiler to compiler. In the above example, we have considered GCC 64 bit.

1.6.6 Control Structures

The method of achieving the objectives of an accurate, error-resistant, and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure
2. Selection structure
3. Loop structure

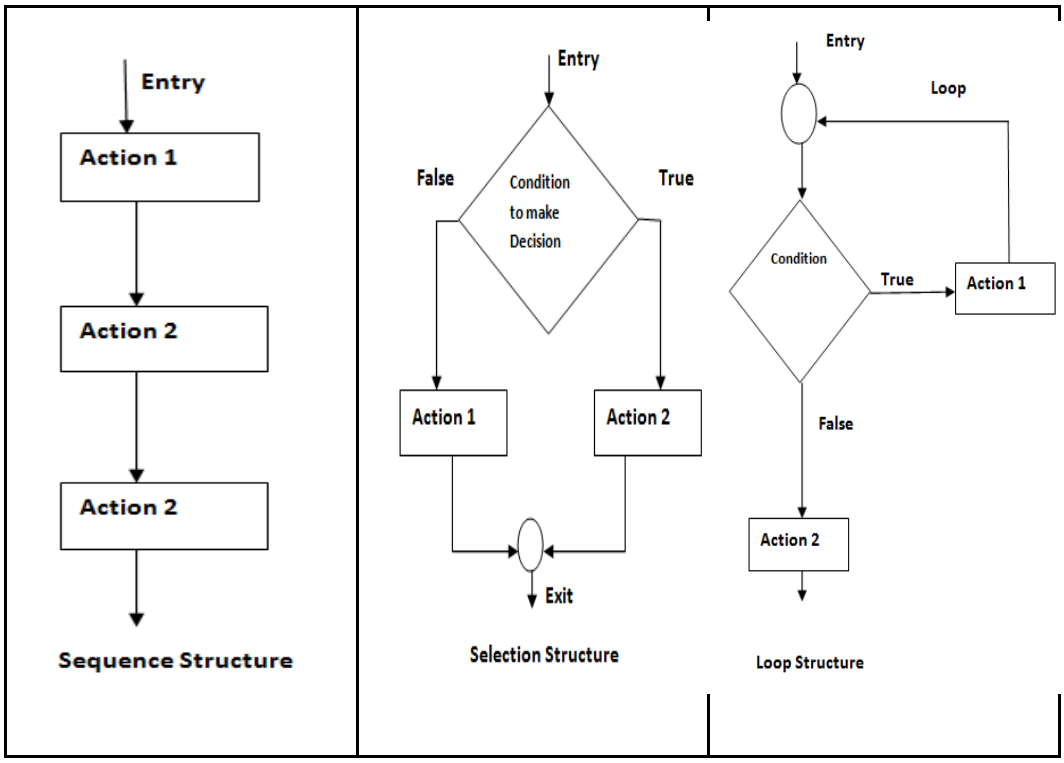


Fig.4 Control Structures in C++

1. Sequence structure

The sequence structure is built into C++. In C++ statements are executed one after the other in the order in which they are written that is, in sequence. For example, a typical sequence structure in which two calculations are performed in order.

Total=Total+grade; //add a grade to total

Counter=Counter+1; // add a 1 to counter

2. Selection structure:

This structure used for decisions, branching choosing between 2 or more alternative paths. In C++, following are the types of selection statements:

- if
- if/else
- switch

if a condition is fulfilled then only execute an instruction or block of instructions. Its form is:

Syntax:

if (*condition***)** *statement*

where *the condition* is the expression that is being evaluated. If this condition is **true**, ***the statement*** is executed. If it is false, the *statement* is ignored (not executed) and the program continues on the next instruction after the conditional structure.

For example, the following code fragment prints out **x is 200** only if the value stored in variable **x** is indeed 200:

Example:

```
if (x == 20)
  cout << "x is 20";
```

If we want more than a single instruction to be executed in case that *condition* is **true** we can specify a *block of instructions* using curly brackets { }:

```
if (x == 20)
{
  cout << "x is ";
  cout << x;
}
```

We can additionally specify what we want that happens if the condition is not fulfilled by using the keyword else. Its form used in conjunction with it is:

Syntax:

if (*condition***)** *statement1* **else** *statement2*

Example:

```
if (x == 98)
    cou                cout << "x is 98";
else
    cout << "x is not 98";
```

prints out on the screen x are 99 if indeed x is worth 99, but if it is not -and only if not- it prints out x is not 100. The *if + else* structures can be concatenated to verify a range of values. The following example shows its use telling if the present value stored in **x** is positive, negative or none of the previous, that is to say, equal to zero.

Example

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

***switch*:**

The syntax of the *switch* instruction is a bit peculiar. Its objective is to check several possible constant values for expression, something similar to what we did at the beginning of this section with the linking of several *if* and *else if* sentences. Its form is the following:

Syntax:

```
switch (expression) {
  case constant1:
    block of instructions 1
    break;
  case constant2:
    block of instructions 2
    break;
  .
  .
  .
  default:
    default block of instructions
  }
```

Example:

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

3. **Loop Structure (Repetition)**: used for looping, in details, repeating a piece of code multiple times in a row. In C++, there are three types of loops:
   - while
   - do/while
   - for

The *while* loop.

Its format is:

while (*expression***)** *statement*

and its function is simply to repeat *statement*s while *expression* is true.

For example, we are going to build a program to count down using a *while* loop:

Example:

```
// custom countdown using while
#include <iostream.h>
int main ()
{
 int n;
 cout << "Enter the starting number > ";
 cin >> n;
 while (n>0) {
  cout << n << ", ";
  --n;
 }
 cout << "FINISH!";
 return 0;
}
```

The *do-while* loop.

Format:

do *statement* **while (***condition***);**

Its functionality is the same as the *while* loop except for that *condition* in the *do-while* is evaluated after the execution of *statement* instead of before, granting at least one execution of *statement* even if the *condition* is never fulfilled. For example, the following program echoes any number you enter until you enter 0.

Example:

```
// number echoer
#include <iostream.h>
int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

The *for* a loop.

Its format is:

for (*initialization***;** *condition***;** *increase***)** *statement*;

and its main function is to repeat *statement*s while the *condition* remains true, like the *while* loop. But **also, to** provide places to specify *initialization* instruction and an *increase* in instruction. So this loop is specially designed to perform a repetitive action with a counter.

It works the following way:

1, *initialization* is executed. Generally, it is an initial value set for a counter variable. This is executed only once.

2, *the condition* is checked, if it is **true** the loop continues, otherwise the loop finishes, and the *statement* is skipped.

3, the *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets **{ }**.

4, finally, whatever is specified in the *increase*d field is executed and the loop gets back to step 2.

Example:

```
// countdown using a for loop
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}
```

1.6.7 Operators and Expressions

An operator is a symbol or sign that tells the compiler to perform mathematical or logical manipulations. C++ provide the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

There are following arithmetic operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then –

Table.4 Arithmetic Operators

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and the remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Example:

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

main() {
   int a = 21;
   int b = 10;
   int c ;
   c = a + b;
   cout << "1 - Value of c is :" << c << endl ;
   c = a - b;
   cout << "2 - Value of c is  :" << c << endl
   ;
   c = a * b;
   cout << "3 - Value of c is :" << c << endl ;
   c = a / b;
   cout << "4 - Value of c is  :" << c << endl ;

   c = a % b;
   cout << "5 - Value of c is  :" << c << endl ;
   c = a++;
   cout << "6 - Value of c is :" << c << endl ;
   c = a--;
   cout << “7 - Value of c is  :" << c << endl ;
   return 0;
}
```

Output:

```
1 - Value of c is  :31
2 - Value of c is  :11
3 - Value of c is  :210
3 - Value of c is  :210
4 - Value of c is  :2
5 - Value of c is  :1
```

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then –

Table.5 Relational Operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then the condition becomes true.	(A != B) is true.
>	Checks if the value of the left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of the left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of the left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of the left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example:

```
#include <iostream>
using namespace std;
main()
{
  int a = 25;
  int b = 78;
  int c ;
  c = a + b;
  cout << "1 - Value of c is :" << c << endl ;
  c = a - b;
  cout << "2 - Value of c is  :" << c << endl ;
  c = a * b;
  cout << "3 - Value of c is :" << c << endl ;
  c = a / b;
  cout << "4 - Value of c is  :" << c << endl ;
  c = a % b;
  cout << "5 - Value of c is  :" << c << endl ;
  return 0;
}
```

Output:

```
1 - Value of c is :103
2 - Value of c is  :-53
3 - Value of c is :1950
4 - Value of c is  :0
5 - Value of c is  :25
```

Logical Operators

There are following logical operators supported by the C++ language.

Assume variable A holds 1 and variable B holds 0, then –

Table 6: Logical Operators

Operator	Description	Example
&&	Called Logical **AND operator**. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical **OR Operator**. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical **NOT Operator**. Use to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make false.	!(A && B) is true.

Example:

```
#include <iostream>
using namespace std;
main()
{
  int a = 50;
  int b = 2;
  int c ;
  if(a && b)
```

```
  {
    cout << "1 - Condition is true"<< endl ;
  }
  if(a || b)
  {
    cout << "2 - Condition is true"<< endl ;
  }
  /* After changing the values of  a and b variable */
  a = 4;
  b = 60;
  if(a && b)
  {
    cout << "3 - Condition is true"<< endl ;
  } else
  {
    cout << "4 - Condition is not true"<< endl ;
  }
  if(!(a && b))
  {
    cout << "5 - Condition is true"<< endl ;
  }
  return 0;
}
```

Output:

```
1 - Condition is true
2 - Condition is true
3 - Condition is true
```

Bitwise Operators

Bitwise operator works on bits and performs a bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Table 7: Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Example:

```
#include <iostream>
using namespace std;
main()
{
  unsigned int a = 98;
  unsigned int b = 28;
  int c = 0;
  c = a & b;
  cout << "1 - Value of c is : " << c << endl ;
  c = a | b;
  cout << "2 - Value of c is: " << c << endl ;
  c = a ^ b;
  cout << "3 - Value of c is: " << c << endl ;
  c = ~a;
  cout << "4 - Value of c is: " << c << endl ;
  c = a << 2;
  cout << "5 - Value of c is: " << c << endl ;
  c = a >> 2;
  cout << "6 - Value of c is: " << c << endl ;
  return 0;
}
```

Output:

```
1 - Value of c is : 0
2 - Value of c is: 127
3 - Value of c is: 127
4 - Value of c is: -99
5 - Value of c is: 392
6 - Value of c is: 24
```

Assignment Operators

There are following assignment operators supported by C++ language –

Table 8: Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example:

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

int main()
{
   // using "=" operator
   int a = 10;
   cout << "Value of a is "<<a<<"\n";                  // using "+=" operator
   a += 10;
   cout << "Value of a is "<<a<<"\n";                 // using "-=" operator
   a -= 10;
   cout << "Value of a is "<<a<<"\n";                 // using "*=" operator
   a *= 10;
   cout << "Value of a is "<<a<<"\n";                // using "/=" operator
   a /= 10;
   cout << "Value of a is "<<a<<"\n";

   return 0;
}
output:
Value of a is 10
Value of a is 20
Value of a is 10
Value of a is 100
Value of a is 10
```

1.7 **Summary**

- C++ is a superset of the C language.
- C++ adds a number of object-oriented features.
- C++ supports interactive input & output features.
- Object-Oriented Programming was invented to overcome the drawback of the Procedural Oriented programming.
- C++ provides various types of tokens that include keywords, identifiers, constants, string, and pointers.
- C++ provides various applications which is use in real life problems.

1.8 Reference for further reading

Reference Books:

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition,Bjarne Stroustrup, Addison Wesley

Web References:

1. www.geeksforgeeks.org
2. www.javatpoint.com

1.9 Unit End Exercises

1. What are Programming Paradigms? Explain the different type of Programming Paradigms.
2. What is the basic structure of a C++ program?
3. Explain compilation and execution flow of c program?
4. What are preprocessor directives in C++?
5. What are the features of Object-Oriented Programming?
6. What is the application of C++?
7. Explain difference between C & C++?

❖❖❖

UNIT 2

2

INTRODUCTION TO C++

Unit Structure

2.0 Objectives

After the completion of this unit, you will be able to understand following things,

1. Structure of C++ program & flow of the program.
2. Classes & objects, access modifiers, data members, member functions
3. Passing parameters to a function in different forms
4. Function overloading

2.1 Introduction

As a programmer, it is very important to know the structure & flow of the program. In this unit, we are going to learn about the structure, classes, objects, member functions, data members, access modifiers. In the end, learner will be able to create a class, can declare data members, member functions, can update the accessibility using access modifiers, can access data from the class using objects.

2.2 Structure of the C++ Program

Let us look at the very basic C++ Program. The name of the program is "Demo" so its source file is "Demo.CPP". It simply displays a sentence on the screen.

```
#include<iostream.h>
Using namespace std;
int main()
{
cout<<"Welcome to the world of C++\n";
return(0);
}
```

Let's examine the structure of a C++ program in detail.

1. Pre-processor directive gives instruction to the compiler. So *#include* is the pre-processor directive used in above code.
2. Followed by a pre-processor, there comes header file. <iostream.h> is the header file used in above program.
3. Function: Function is the one of the most important building blocks of C++. The above program contains only one function, main().
4. The parenthesis (), after the "main" indicates that it's a function and not a variable.
5. The word "int" preceding the main() function indicates that the particular function has a return value of type *int*.
6. The Body of the function is initialized by opening braces (curly brackets) and ends with closing braces.

7. Opening & closing braces of a C++ program signifies start & the end of the function body respectively.
8. The function body can contain multiple statements. Program statement is a fundamental unit of C++ program. Each statement ends with a semicolon (;).
9. "Cout" is called an identifier which corresponds to standard output stream. The operator "<<" is called the insertion or put to operator.

2.3 Execution Flow

When we run a program or execute a program, execution always starts from the function called main(). If there is no main() function in your program, you will get an error as the compiler will not understand where to start the execution. The main() function calls member functions in various objects to carry out the program's real work. It also contains calls to other standalone functions.

The following flowchart shows the execution flow of a C++ Program:

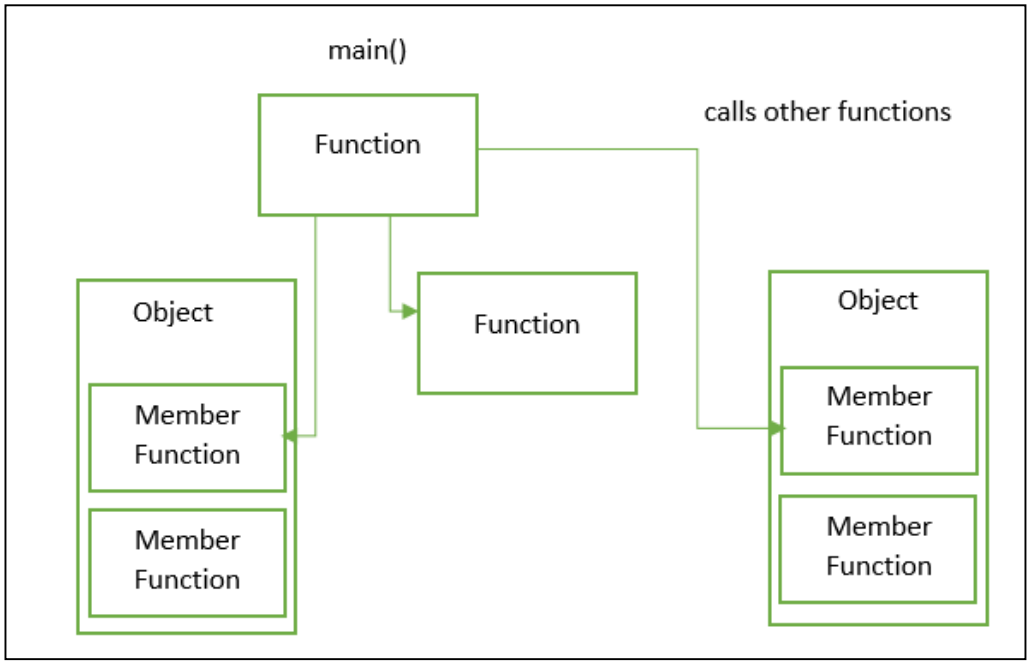


Fig shows flow of a C++ Program

2.4 Classes & Objects

There are various categories of objects that forms the basis for Object Oriented Programming. Following are some categories of objects.

1. Physical objects
    a. Automobiles in a traffic-flow simulation
    b. Electrical components in a circuit-design program
    c. Countries in an economics model
    d. Aircraft in an air traffic control system
2. Elements of the computer-user environment
    a. Windows
    b. Menus
    c. Graphics objects (lines, rectangles, circles)
    d. The mouse, keyboard, disk drives, printer
3. Data-storage constructs
    a. Customized arrays
    b. Stacks
    c. Linked lists
    d. Binary trees
4. Human entities
    a. Employees
    b. Students
    c. Customers
    d. Salespeople
5. Collections of data
    a. An inventory
    b. A personnel file
    c. A dictionary
    d. A table of the latitudes and longitudes of world cities
6. User-defined data types
    a. Time
    b. Angles
    c. Complex numbers
    d. Points on the plane
    e. Physical Objects

7. Components in computer games
    a. Cars in an auto race
    b. Positions in a board game (chess, checkers)
    c. Animals in an ecological simulation
    d. Opponents and friends in adventure games

The match between programming objects and real world objects is the result of combining data and functions.

Let's consider a basic program as shown below,

```
// Demonstrate small C++ Program working with objects
#include <iostream>
using namespace std;
class Demo//define a class
{
private:
int data; //class data
public:
void setdata(int d) //member function to set data
{
data = d;
}
void showdata() //member function to display data
{
    cout << "Data is "<<data << endl; }
};
int main()
{
Demo  s1, s2; //define two objects of class Demo
s1.setdata(10); //call member function to set data
s2.setdata(15);
s1.showdata(); //call member function to display data
```

```
s2.showdata();
return 0;
}
```

Program #1

In the above program Class Demo is being created. The Class Demo has one data item and two member functions. We can access the data item inside the Demo class using the available member functions. The first member function sets the value for the data item and the other member function displays the value of the data item.

Placing data & and its functions together into a single entity is a central idea of Object Oriented Programming. So we can show it diagrammatically as follows,

Class

Figure shows data & functions in a class

A class is thus a description of a number of similar objects. This fits our non-technical understandingof the word class. Mango. apple and banana are members of the Fruit_basket class. There is no one fruit called “Fruit,” but specific fruits with specific names are members of this class if they possess certain characteristics. An object is often called an “instance” of a class.

Defining Objects:

Object is called as an instance of a class. Let’s check the output of the above program. We have observed that two objects named s1 & s2 are used to access the class. Each of the two objects is given a value, and each displays its value. Here’s the output:

Data is 10 <---------- Object s1 displayed this

Data is 15 <---------- Object s2 displayed this

In the above program, inside main() function, we see two objects s1, s2. The statement

Demo s1, s2;

means objects of type "Demo" have been created. The same process is also called as **instantiating**. The above statement also states that the structure of object. It only describes how they will look when they are created, just as a structure definition describes how a structure will look but doesn't create any structure variables. It is objects that participate in program operations.

Defining a class:

```
class demo2 //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
{ somedata = d; }
void showdata() //member function to display data
{ cout << "\nData is " << somedata; }
};
```

Program #2

The definition starts with the keyword CLASS, followed by the class name. Like a structure, the body of the class is delimited by braces and terminated by a semicolon. The above example of a class also shows keywords, Private & public. This is a feature of Object Oriented programming called as 'data hiding'.

Class data:

The above class demo2 contains one data item with 'int' datatype. The data item is also called as data members. There can be any number of data items in a class. We can also set the visibility of the data members as private or public.

2.5 Member Functions

Member functions are functions that are included within a class. These are the functions which are included within the class. Setdata() & showdata() are the member functions used in class demo2. The function bodies of these functions have been written on the same line as the braces that delimit them. You could also use the more traditional format for these function definitions:

```
void setdata (int d)
{
    somedata = d;
}
 And
void showdata()
{
    cout << "\nData is " << somedata;
};
```

The member functions in the **Demo2** class perform operations that are quite common in classes: setting and retrieving the data stored in the class. The setdata() function accepts a value as a parameter and sets the somedata variable to this value. The showdata() function displays the value stored in somedata. Member functions defined inside a class this way are created as inline functions by default.

Calling Member Functions:

In the above Program #1, the below statements from main() function calls function setdata().

s1.setdata(10);

s2.setdata(15);

These statements don't look like normal function calls because the object names s1 and s2 connected to the function names. This strange syntax is used to call a member function that is associated with a specific object. Because setdata() is a member function of the Demo 1 class, it must always be called in connection with an object of this class.

A member function is always called to act on a specific object, not on the class in general.

Member functions of a class can be accessed only by an object of that class. To use a member function, the dot operator (the period) connects the object name and the member function. The syntax is similar to the way we refer to structure members, but the parentheses signal that we're executing a member function rather than referring to a data item. The dot operator is also called the class member access operator.

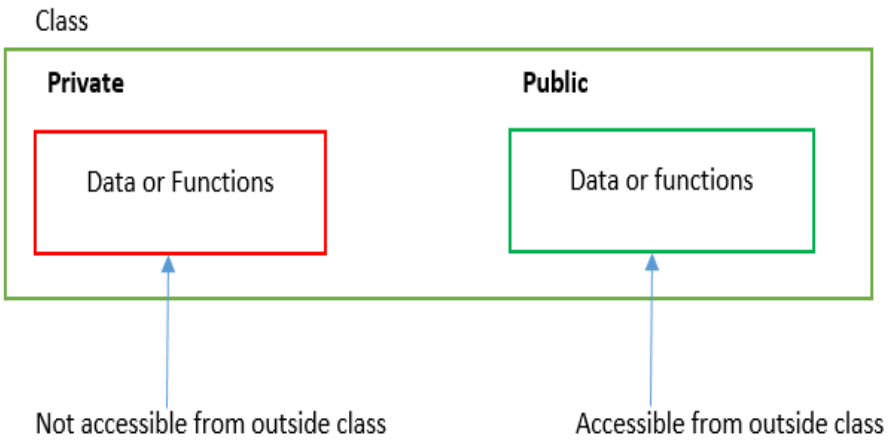
2.6 Access Modifiers

A key feature of object-oriented programming is data hiding. This term does not refer to the activities of particularly paranoid programmers; rather it means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class.

The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. The primary mechanism of hiding data is to make the function 'private'. Private data or functions can be accessed only inside the same class. On the other hand, public data and member functions can be accessed from anywhere.

What is the importance of using Access modifier in a program? In some cases, it makes sense for objects of the derived class to access the public functions of the base class if they want to perform a basic operation, and to access functions in the derived class to perform the more specialized operations that the derived class provides.

In such cases public derivation is appropriate. In some situations, however, the derived class is created as a way of completely modifying the operation of the base class, hiding or disguising its original interface. The diagram below shows the accessibility of Private & public data or functions in a class.

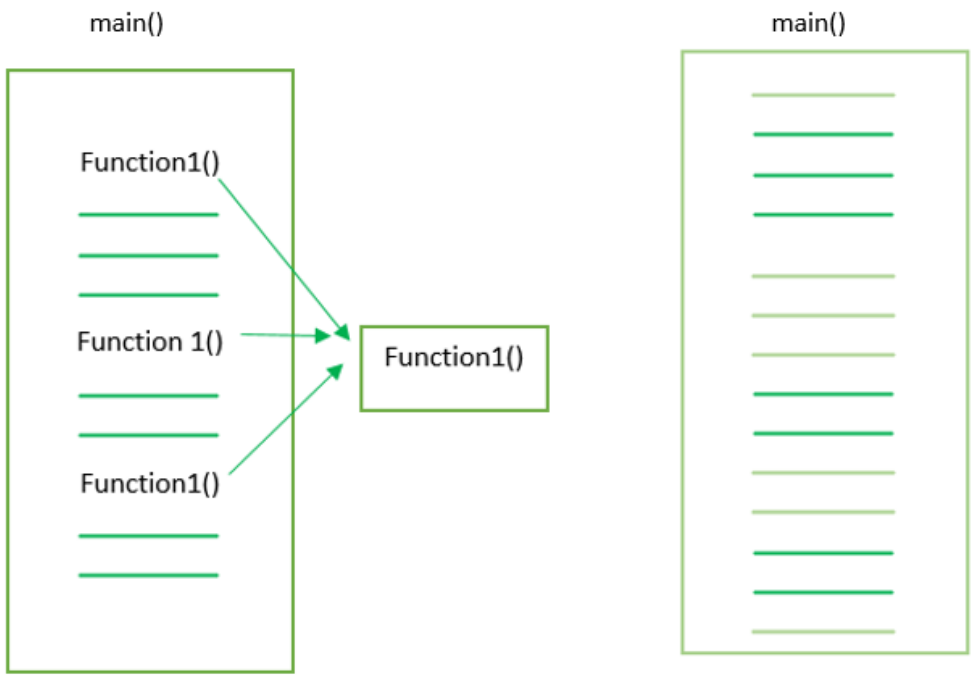


2.7 Inline Functions

We all know that use of functions saves memory space but the whole process of calling a function and jumping back to the next instruction in the program is time consuming. Hence Inline functions came in picture. To save execution time in short functions, you may elect to put the code in the function body directly inline with the code in the calling program. That is, each time there's a function call in the source file, the actual code from the function is inserted, instead of a jump to the function.

Long sections of repeated code are generally better off as normal functions: The savings in memory space is worth the comparatively small sacrifice in execution speed. But making a short section of code into an ordinary function may result in little savings in memory space, while imposing just as much time penalty as a larger function.

In fact, if a function is very short, the instructions necessary to call it may take up as much space as the instructions within the function body, so that there is not only a time penalty but a space penalty as well. In such cases you could simply repeat the necessary code in your program, inserting the same group of statements wherever it was needed. The trouble with repeatedly inserting the same code is that you lose the benefits of program organization and clarity that come with using functions. The program may run faster and take less space, but the listing is longer and more complex.



(a) Normal Function (b) Code with inline functions

For Example:

//Inliner example

```
#include <iostream>
using namespace std;
// converts kilograms to grams
inline float kg2gm (int kilo)
{
return 1000 * kilo;
}
//------------------------------------------------------------
int main()
{
float kg;
cout << "\nEnter kilograms: ";
cin >> kg;
```

```
cout << “Kilograms to grams “ <<kg2gm(kg)
<< endl;
return 0;
}
```

In the above program, we declared and defined function kg2gm using the keyword inline. Later in the main(), we have called the inline function and calculated Kilograms to grams.

2.8 Passing parameters to a Function

Just like passing constants to functions, the function gives these new variables the names and data types of the parameters specified in the declarator: ch of type char and n of type int. It initializes these parameters to the values passed. They are then accessed like other variables by statements in the function body. Passing arguments in this way, where the function creates copies of the arguments passed to it, is called passing by value.

2.8.1 Pass by value

In call by value, the actual value that is passed as argument is not changed after performing some operation on it. When call by value is used, it creates a copy of that variable into the stack section in memory. When the value is changed, it changes the value of that copy, the actual value remains the same.

Example:

```
#include<iostream>
void my_fun(int x)
{
  x = 50;
  cout<< "Value of x from my_fun: " << x << endl;
}
main()
{
  int x = 10;
  my_fun(x);
  cout << "Value of x from main function: " << x;
```

```
}
Output:
Value of x from my_fun: 50
Value of x from main function: 10
```

2.8.2 Pass by Reference

In call by reference the actual value that is passed as argument is changed after performing some operation on it. When call by reference is used, it creates a copy of the reference of that variable into the stack section in memory. Is uses a reference to get the value. So when the value is changed using the reference it changes the value of the actual variable.

```
Example:
#include<iostream>
void my_fun(int &x)
{
  x = 50;
  cout << "Value of x from my_fun: " << x << endl;
}
main() {
  int x = 10;
  my_fun(x);
  cout << "Value of x from main function: " << x;
}
Output:
Value of x from my_fun: 50
Value of x from main function: 50
```

2.9 Function with default arguments

An argument is a piece of data (an int value, for example) passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling

it. A function without arguments will not work. This can be better understood with following program. The following program uses three different functions with same name to handle different number of arguments.

```
#include <iostream>
using namespace std;
void repchar(char='*', int=45); //declaration with
                                //default arguments
int main()
{
repchar();              //prints 45 asterisks
repchar('=');           //prints 45 equal signs
repchar('+', 30);   //prints 30 plus signs
return 0;
}
//-------------------------------------------------------
// repchar()
// displays line of characters
void repchar(char ch, int n) //defaults supplied
{
for(int j=0; j<n; j++) //loops n times
cout << ch; //prints ch
cout << endl;
}
```

In this program the function repchar() takes two arguments. It's called three times from main(). The first time it's called with no arguments, the second time with one, and the third time with two. Why do the first two calls work? Because the called function provides default arguments, which will be used if the calling program doesn't supply them. The default arguments are specified in the declaration for repchar():

```
void repchar(char='*', int=45); //declaration
```

The default argument follows an equal sign, which is placed directly after the type name. You

can also use variable names, as in

void repchar(char reptChar='*', int numberReps=45);

If one argument is missing when the function is called, it is assumed to be the last argument. The repchar() function assigns the value of the single argument to the ch parameter and uses the default value 45 for the n parameter. If both arguments are missing, the function assigns the default value '*' to ch and the default value 45 to n. Thus the three calls to the function all work, even though each has a different number of arguments.

2.10 Function Overloading

An overloaded function appears to perform different activities depending on the kind of data sent to it. It performs one operation on one kind of data but another operation on a different kind. Let's clarify with following example.

```
#include <iostream>
using namespace std;
void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}
int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Ref. Geeksforgeeks

In the above program, we can see print function thrice. However, the function print()looks different in all three cases. No. of parameters used in each function is different. Whenever we call print function and pass parameters or arguments, the respective function is being called and executed.

2.11 Object as a Parameter

we know that, we can pass any type of arguments within the member function and there are any numbers of arguments. In C++ programming language, we can also pass an object as an argument within the member function of class. This is useful, when we want to initialize all data members of an object with another object, we can pass objects and assign the values of supplied object to the current object. For complex or large projects, we need to use objects as an argument or parameter.

Example:

```
#include <iostream>
using namespace std;
class Demo
{
    private:
        int a;
    public:
        void set(int x)
        {
            a = x;
        }
        void sum(Demo ob1, Demo ob2)
        {
            a  = ob1.a + ob2.a;
        }
        void print()
```

```
        {
            cout<<"Value of A :  "<<a<<endl;
        }
};
int main()
{
    //object declarations
    Demo d1;
    Demo d2;
    Demo d3;
    //assigning values to the data member of objects
    d1.set(10);
    d2.set(20);
    //passing object d1 and d2
    d3.sum(d1,d2);

    //printing the values
    d1.print();
    d2.print();
    d3.print();
    return 0;
}
```

Output:

Value of A: 10

Value of A: 20

Value of A: 30

2.12 Static data members & Functions

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by re-declaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Example:

```
#include <iostream>
class Box
 {
   public:
      static int objectCount;
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
 // Increase every time object is created
         objectCount++;
      }
 double Volume()
{
return length * breadth * height;
}
private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
```

```
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;
return 0;
}
Output:
Constructor called.
Constructor called.
Total objects: 2
```

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

A static member function can only access static data member, other static member functions and any other functions from outside the class. Static member functions have a class scope and they do not have access to the this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

```
Example:
include <iostream>
class Box
{
  public:
     static int objectCount;
     // Constructor definition
```

```
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }
    static int getCount() {
      return objectCount;
    }
private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
  // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
  // Print total number of objects after creating object.
  cout << "Final Stage Count: " << Box::getCount() << endl;
  return 0;
}
```

```
Output:
Initial Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

2.13 Let us sum up

1. Classes includes objects, member functions, data members, access modifiers.
2. Object can be passed using value & reference.
3. Function overloading can be done using the same function name using different parameters.

2.14 List of References

1. The Complete Reference C, 4th EditionHerbert Sehlidt,Tata Mcgraw Hill
2. Object Oriented Programming in C++,4th Edition,Robert Lafore,SAMS Techmedia

Web references:

1. https://dev.mysql.com
2. www.github.com
3. Geeksforgeek.com

2.15 Bibliography

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition,BjarneStroustrup,AddisonWesly
3. Starting Out with C++ Early Objects,8th Edition,Tony Gaddis et al,Addison-Wesley
4. C++ How to Program,8th Edition,Deitel and Deitel, Prentice Hall
5. Practical C++ Programming,2nd Edition,Steve Quoaline,O'reilly Publication
6. Absolute C++,4th Edition, Walter Savitch,Pearson Education

2.16 Unit End Exercise

Q1. What is a class? Explain with an example.

Q2. What are access modifiers? How to implement it?

Q3. What is object? Explain any 3 types.

Q4. What is function overloading? Explain with a simple example.

Q5. Difference between pass by value & pass by reference.

❖❖❖

UNIT 2

3

INTRODUCTION TO C++ - CONSTRUCTOR & ARRAY

Unit Structure

3.0 Objectives

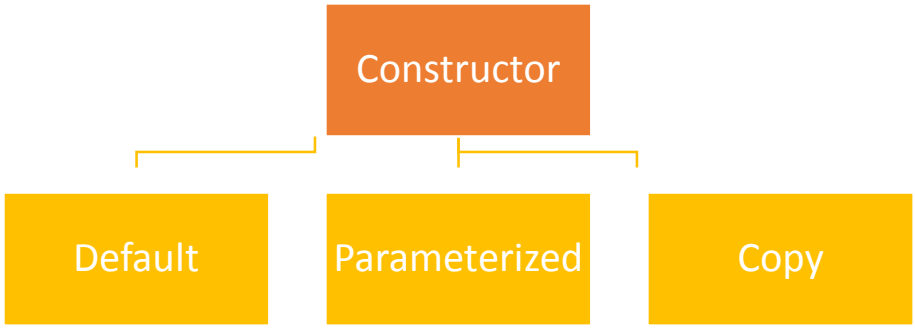
After the completion of this unit, you will be able to understand following things,

1. Constructor, its functions and its types.
2. Introduction to array & types of array.

3.2 Introduction - Constructor

Definition:We know that an object can initialize itself when it's first created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a constructor. A constructor is a member function that is executed automatically whenever an object is created.

There are 3 types of constructor, default, parameterized & copy.



A constructor is different from normal function in following ways:

1. Constructor has same name as the class itself.
2. Constructors don't have return type.
3. A constructor is automatically called when an object is created.
4. If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

3.2.1 Default Constructor

Syntax: The name of the constructor is same the name of function. Default constructor doesn't take any argument, that means it has no parameters.The syntax is more or less same all the three, let us have a look at Default constructor with following example.

Example:

```
#include<iostream>
class construct {
public:
   int a, b;
```

```
    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
  cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
Output:
a: 10
b: 20
```

In the above program, Function ‘construct’ has been defined with a default constructor named ‘construct’. Two values of ‘a’ and ‘b’ have been defined inside the function. This means if user doesn’t pass any value, default constructor would be called automatically. In the main() method we can see, once the object created for the function construct, values from the default constructor are called and displayed as the output.

3.2.2Parameterized Constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

Syntax: The syntax is same as default constructor but with parameters. So the name of the constructor will remain same as per the logic, we will add parameters to it.

Example:

```
#include <iostream>
class Point {
private:
   int x, y;
public:
   // Parameterized Constructor
   Point (int x1, int y1)
   {
      x = x1;
      y = y1;
   }
  int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};
int main()
{
   // Constructor called
   Point p1(10, 15);
   // Access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
```

```
return 0; }
Output:
    P1.x = 10, P1.y = 15
```

In the above program, we can see the use of Parameterized constructor. The name of the parameterised constructor is same as the class name and it has 2 parameters. Once called in main() method using an object with parameters, the call is made to the parameterised constructor and values are displayed accordingly.

3.2.3 Copy Constructor

It is a member function which initializes an object using another object of the same class. Whenever we define one or more non-default constructors (with parameters) for a class, a default constructor (without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Syntax:

```
ClassName (const ClassName &old_obj)
Example:
#include<iostream>
class Point
{
private:
   int x, y;
public:
   Point (int x1, int y1) { x = x1; y = y1; }
   // Copy constructor
   Point (const Point &p2) {x = p2.x; y = p2.y; }
   int getX()          {  return x; }
   int getY()          {  return y; }
};
int main()
```

```
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; // Copy constructor is called here
// Let us access values assigned by constructors
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
   cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
   return 0;
}
```

Output:

p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15

3.2.4 Constructor Overloading

In C++, we can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

Things to remember!

1. Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
2. A constructor is called depending upon the number and type of arguments passed.
3. While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Example:

```
#include <iostream>
class construct
{
public:
   float area;
   // Constructor with no parameters
```

```
    construct()
    {
        area = 0;
    }
    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }
    void disp()
    {
        cout<< area<< endl;
    }
};
int main()
{
    construct o;
    construct o2(10, 20);
    o.disp();
    o2.disp();
    return 1;
}
Output: 0
200
```

In the above example, two constructors were declared and defined. 1 with no parameters & other with parameters. You already know that the constructor without parameter is the default constructor whereas the other constructor is parameterized. When the objects are created and are called in main() method, the object with no parameter will point to default constructor and the other object with parameters will

point to parameterized constructor. So by keeping the same name we can achieve constructor overloading.

3.2.5 Destructor

As constructor is defined to create and execute the program using object, destructor is used to delete an object.

A destructor function is called automatically when the object goes out of scope:

1. the function ends
2. the program ends
3. a block containing local variables ends
4. a delete operator is called

Remember destructor neither take any argument nor return anything. It can be used using tilde(~) sign.

3.3 Array - Introduction

Consider a scenario where we need to find out the average of 50 integer numbers entered by user. We can do this in two ways in C:

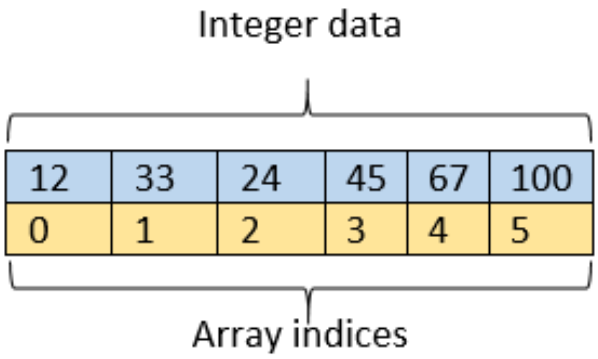
1) Declare & define 50 variables with integer data type and then perform 50 scanf() operations to store the entered values in the variables and then at last calculate the average of them.

2) Or have a single integer array to store all the values, loop the array to store all the entered values in array and later calculate the average.

   Of course the second solution is convenient because it will not only reduce the number of Lines of coding but also its easy to store data of similar data type.

   - An array is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.
   - They are used to store similar type of elements as in the data type must be the same for all elements.
   - They can be used to store collection of primitive data types such as int, float, double, char or they can be any user-defined types such as structures and objects.

- Arrays are like structures in that they both group a number of items into a larger unit. But while a structure usually groups items of different types, an array groups items of the same type.
- The items in a structure are accessed by name, while those in an array are accessed by an index number. Using an index number to specify an item allows easy access to a large number of items.
- For example, if we say declare an array of 6 elements of integer datatype then it will look this,



Following observations can be made from above array.

1. The size of Array= 6
2. First Index= 0
3. Last Index= 5

Example:

```
#include <iostream>
int main()
{
int age[4];
for(int j=0; j<4; j++) //get 4 ages
{
cout << "Enter an age: ";
cin >> age[j]; //access array element
}
```

```
for(j=0; j<4; j++) //display 4 ages
cout << "You entered " << age[j] << endl;
return 0;
}
Output:
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68
You entered 44
You entered 16
You entered 23
You entered 68
```

3.3.1 Array as a class member

Arrays can be used as data items in classes. Let's take an example of a common computer data structure: the stack.

- A stack works like the spring-loaded devices that hold trays in cafeterias. When you put a tray on top, the stack sinks down a little; when you take a tray off, it pops up.
- The last tray placed on the stack is always the first tray removed.
- Stacks are one of the cornerstones of the architecture of the microprocessors used in most modern computers.
- Software stacks offer a useful storage device in certain programming situations.

```
Program:
#include <iostream>
class Stack
{
private:
```

```
enum { MAX = 10 }; //(non-standard syntax)
int st[MAX]; //stack: array of integers
int top; //number of top of stack
public:
Stack() //constructor
{ top = 0; }
void push(int var) //put number on stack
{ st[++top] = var; }
int pop() //take number off stack
{ return st[top--]; }
};
int main()
{
Stack s1;
s1.push(11);
s1.push(22);
cout << "1: " << s1.pop() << endl; //22
cout << "2: " << s1.pop() << endl; //11
s1.push(33);
s1.push(44);
s1.push(55);
s1.push(66);
cout << "3: " << s1.pop() << endl; //66
cout << "4: " << s1.pop() << endl; //55
cout << "5: " << s1.pop() << endl; //44
cout << "6: " << s1.pop() << endl; //33
return 0;
}
```

In the above program, when an item is added to the stack, the index in top is incremented to point to the new top of the stack. When an item is removed, the index in top is decremented. To place an item on the stack—a process called pushing the item—you call the push() member function with the value to be stored as an argument. To retrieve (or pop) an item from the stack, you use the pop() member function, which returns the value of the item.

The main() method in above program exercises the stack class by creating an object, s1, of the class. It pushes two items onto the stack, and pops them off and displays them. Then it pushes four more items onto the stack, and pops them off and displays them.

```
Output:
1: 22
2: 11
3: 66
4: 55
5: 44
6: 33
```

3.3.2Array as objects

We've seen how an object can contain an array. We can also reverse that situation and create an array of objects.

```
Program:
#include <iostream>
class Distance
{
private:
int feet;
float inches;
public:
void getdist() //get length from user
{
cout << "\n Enter feet: "; cin >> feet;
```

```
cout << " Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
};
int main()
{
Distance dist[100]; //array of distances
int n=0; //count the entries
char ans; //user response ('y' or 'n')
cout << endl;
do
{ //get distances from user
cout << "Enter distance number " << n+1;
dist[n++].getdist(); //store distance in array
cout << "Enter another (y/n)?: ";
cin >> ans;
} while( ans != 'n' ); //quit if user types 'n'
for(int j=0; j<n; j++) //display all distances
{
cout << "\nDistance number " << j+1 << " is ";
dist[j].showdist();
}
cout << endl;
return 0;
}
```

Output:

```
Enter distance number 1
Enter feet: 5
Enter inches: 4
```

```
Enter another (y/n)? y
Enter distance number 2
Enter feet: 6
Enter inches: 2.5
Enter another (y/n)? y
Enter distance number 3
Enter feet: 5
Enter inches: 10.75
Enter another (y/n)? n
Distance number 1 is 5'-4"
Distance number 2 is 6'-2.5"
Distance number 3 is 5'-10.75"
```

In this program the user types in as many distances as desired. After each distance is entered, the program asks if the user desires to enter another. If not, it terminates, and displays all the distances entered so far.

3.3.3Strings – Cstyle strings

We noted at the beginning of this chapter that two kinds of strings are commonly used in C++: C-strings and strings that are objects of the string class. We call these strings C-strings, or C-style strings, because they were the only kind of strings available in theC language.

They may also be called char* strings, because they can be represented as pointers to type char. Although strings created with the string class, havesuperseded C-strings in many situations, C-strings are still important for a variety of reasons.First, they are used in many C library functions. Second, they will continue to appear in legacy code for years to come. And third, for students of C++, C-strings are more primitive andtherefore easier to understand on a fundamental level.

3.3.4 String Class

Standard C++ includes a new class called string. This class improves on the traditional C-stringin many ways. For one thing, you no longer need to worry about creating an array of theright size to hold string variables.

The string class assumes all the responsibility for memory management. Also, the string class allows the use of overloaded operators, so you can concatenate string objects with the + operator:

s3 = s1 + s2

There are other benefits as well. This new class is more efficient and safer to use than C-strings were. In most situations it is the preferred approach. In this section we'll examine the string class and its various member functions and operators.

3.3.4.1Defining & Assigning String Objects

You can define a string object in several ways. You can use a constructor with no arguments,creating an empty string. You can also use a one-argument constructor, where the argument is a C-string constant; that is, characters delimited by double quotes. As in our homemade String class, objects of class string can be assigned to one another with a simple assignment operator.

Program:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string s1(“Man”); //initialize
string s2 = “Beast”; //initialize
string s3;
s3 = s1; //assign
cout << “s3 = “ << s3 << endl;
s3 = “Neither “ + s1 + “ nor “; //concatenate
s3 += s2; //concatenate
cout << “s3 = “ << s3 << endl;
s1.swap(s2); //swap s1 and s2
cout << s1 << “ nor “ << s2 << endl;
return 0;
}
```

Output:

s3 = Man

s3 = Neither Man nor Beast

3.4 Let us sum up

1. Classes includes objects, member functions, data members, access modifiers.
2. Object can be passed using value & reference.
3. Function overloading can be done using the same function name using different parameters.

3.5 List of References

1. The Complete Reference C, 4th EditionHerbert Sehlidt,Tata Mcgraw Hill
2. Object Oriented Programming in C++,4th Edition,Robert Lafore,SAMS Techmedia

Web references:

1. https://dev.mysql.com
2. www.github.com
3. Geeksforgeek.com

3.6 Bibliography

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, Addison Wesly
3. Starting Out with C++ Early Objects,8th Edition,Tony Gaddis et al,Addison-Wesley
4. C++ How to Program,8th Edition,Deitel and Deitel, Prentice Hall
5. Practical C++ Programming,2nd Edition,Steve Quoaline,O'reilly Publication
6. Absolute C++,4th Edition, Walter Savitch,Pearson Education

3.7 Unit End Exercise

Q1. What is Constructor? Explain its types.

Q2. Explain the use of Copy Constructor with an example.

Q3. What is Constructor overloading?

Q4. What is an array? Explain with an example.

❖❖❖

UNIT 3

4

OPERATOR OVERLOADING

Unit Structure

4.0 Objective

At the end of this unit, students will be able to:

- Recognize the significance of operator overloading
- Understand the rules for unary operator overloading
- Understand the rules for binary operator overloading
- Know the importance of type conversion operators

4.1 Operator Function-Introduction

C++ offers a rich collection of operators in its basket. Some of those operators we have already covered in the previous units. One of the enriching feature offered by C++ is known as operator overloading. Modern day most of the programming languages makes use of this feature in order to support OOP features.

Overloading an operator simply means providing special syntax and semantics to an existing operator. It enables an operator to exhibit more than one operation simultaneously, as illustrated below:

For example, we all are aware that an addition operator (+) is essentially a numeric operator and therefore, requires two number operands. The significance of this operator is to add the numeric values at either side of operator and compute the summation of the numeric values. Interestingly, the same addition operator (+) cannot be used in adding two strings. However, if we provide special meaning to addition operator (+) we can extend the operation of addition operator to include string concatenation. Consequently, the addition operator would work as follows:

```
string s1="LAP";
string s2="TOP";
string s3=s1+s2;
cout<<s3;
```

Output

LAPTOP

This act of reinventing the effect of an operator is called ***operator overloading***. Please note that the original meaning and action of the operator however remains unchanged. Only an additional meaning is added to it.

Similar to function overloading which allows different functions with different argument list having the same name, an operator overloading can be reinvented to perform additional tasks. Operator overloading is accomplished using a special function, which can be a *member function* or *friend function*.

Syntax:
<return_type> operator <operator_being_overloaded>(<argument list>);

where, operator is the keyword and is preceded by the return_type of the operation.

Note: In order to overload the addition operator (+) to perform the concatenation of two characters, the following declaration, which could be either member or friend function, would be essential

```
char * operator + (char *s2);
```

Let's understand the overloading principle in more detail. In previous chapters we have learn about the overloading principle which was applied to functions which we call as function overloading. The same overloading principle is applied to an operator here. In C++, most of operators can be extended to work with both *built-in* types as well as for *classes*. However, there are few operators which cannot be overloaded which will see in sometime. A programmer can apply his skill and discover a new operator to a class by overloading the built-in operator function to perform some precise computation when the operator is used on objects of that class. The question comes to a mind "*Can operator overloading be implemented for real world problems?"* The answer to that is yes it certainly possible to that, making it very easy to develop and deploy code that feels natural. However in certain situations operator overloading, like any advanced C++ feature, makes the language slightly more complex. One must understand that every operators tend to have very precise meaning, and most programmers don't expect operators to do a lot of versatile work, so overloading operators can be slightly more confusing at times. Our intention in writing this book, is to keep it as simple as possible so that the readers will be able to digest it fairly easily.

4.2 Implementing Operator Overloading

In C++, operator overloading can be achieve by implementing a function which can be:

4.2.1 Member Function

4.2.2 Non-Member Function

4.2.3 Friend Function

In the first case, operator overloading function can be a member function if and only if there exist a condition in which Left operand is an Object of that class. In the second case, if the Left operand is different than the one which we have defined above, then Operator overloading function must be a non-member function. Third case is only possible only when there is a condition when there is a need to access to the private and protected members of class. We will examine each of the above case in detail.

4.2.1 Member Function

- This is the first widely used and the most popular method of implementing operator overloading concept.
- While overloading the operator by using this technique, the following points we need to consider
    i. The overloaded operator must be added as a member function of the left operand.
    ii. The left operand becomes the implicit *this object
    iii. All other operands become function parameters.
- Let us look at the example which will make the understanding slightly more simpler.

```
#include <iostream>
class ABC
{
private:
   int m_abc;

public:
   ABC(int abc) { m_abc = abc; }

   // Overload Cents + int
   ABC operator +(int value);

   int getABC() const { return m_abc; }
};

// note: this function is a member function!
ABC ABC::operator +(int value)
{
   return ABC(m_abc + value);
```

```
}


int main()
{
    ABC a1(6);
    ABC a2 = a1 + 2;
    std::cout << "I have " << a2.getABC() << " rupees.\n";
    return 0;
}
```

4.2.2 Non Member Function

- Operator overloading can also be achieved by using non-member function.
- A non-member operator overloading function simply has the right name and does whatever you want.
- For example, suppose we add two BitMoney objects and get a third BitMoney object that has the sum from the first two, etc.
- We define the function named operator+ that takes two arguments of BitMoney type and returns a BitMoney object with the correct values.
- Let's use the example version of BitMoney in which the member variables n,d, etc., are private.

```
BitMoney operator+ (BitMoney lhs, BitMoney rhs)
{
BitMoney sum;
sum.set_n(lhs.get_n () + rhs.get_n () );
... etc ...
return sum;
}
```

- This function creates a new object, gets the n value from the lhs and rhs objects, and sets the n of the new object to their sum.
- Finally it returns the new object.

So now we can write:

x3 = x1 + x2;

- The "x1 + x2" will be compiled into a call to our operator+ function that takes two BitMoney objects as arguments. It returns an object containing the sum, whose values then get copied into x3. The above statements can be rewritten as follows:

x3 = x1 + x2;

x3 = operator+ (x1, x2);

4.2.3 Friend Function

- They offer better flexibility to the class in which they are defined.
- Please note that these functions are not a members of the class and they there is no 'this' pointer concept.
- While overloading unary operator using friend function, you need to pass one argument whereas for binary operator, one need to pass two arguments.
- Private members of a class can be directly accessible by using friend functions.

Syntax:

```
friend return-type operator operator-symbol (var 1, var2)
{
    //Statements;
}
```

Let us understand friend function by implementing a small program by overloading unary operator.

```
#include<iostream>
using namespace std;
class UF
{
    int x=10;
    int y=20;
    int z=30;
```

```
    public:
        void getv()
        {
            cout<<"Values of X, Y & Z\n";
            cout<<x<<"\n"<<y<<"\n"<<z<<"\n"<<endl;
        }
        void display()
        {
            cout<<x<<"\n"<<y<<"\n"<<z<<"\n"<<endl;
        }
        void friend operator -(UF &a);      //Pass by reference
};
void operator-(UF &a)
{
    a.x = -a.x;     //Object name must be used as it is a friend function
    a.b = -a.b;
    a.c = -a.c;
}
int main()
{
    UF a1;
    a1.getv();
    cout<<"Before Unary Overloading\n";
    a1.display();
    cout<<"After Unary Overloading \n";
    -a1;
     a1.display();
     return 0;
}
```

Output:

Values of X, Y & Z

10

20

30

Before Unary Overloading

10

20

30

After Unary Overloading

-10

-20

-30

Analysis of program

In the above program, operator – is overloaded using friend function. The operator() function is defined as a Friend function. The statement -a1 invokes the operator() function. The object a1 is created of class UF. The object itself acts as a source and destination object. This can be accomplished by sending reference of an object. The object a1 is a reference of object a. The values of object a1 are replaced by itself by applying negation.

4.3 Unary Operator Overloading

- Every developer knows the operation overloading concept while implementing it in C++.
- Although the operation looks quite simple to redefine the operators in operator overloading, there are certain restrictions and limitation in overloading the operators.
- Few of them are listed below:
    1. It allows only existing operators to be overloaded. New operators cannot be overloaded.
    2. Every overloaded operator should have at least one operand which is of the type of user defined.

3. Operator overloading can be achieved without changing the basic meaning of an operator. For example, we cannot redefine the minus(-) operator to add one value with the other.

4. One must note that the overloaded operators cannot be overridden and should follow the syntax rules of the original operators.

5. List of operators which cannot be overloaded are
   - size of operator(sizeof),
   - membership operator(.),
   - pointer to member operator(.*),
   - scope resolution operator(::),
   - conditional operators(?:) etc

6. There are certain cases in which we cannot make use of friend functions to overload certain operators. In such cases we can make use of member function to overload them.

   Please note friend functions cannot be used with
   - assignment operator(=),
   - function call operator(()),
   - subscripting operator([]),
   - class member access operator(->) etc.

7. For overloading unary operators, one must make use of a member function, which take no explicit arguments and return no explicit values. Please note that, those operators overloaded by means of a friend function, should take one argument by reference (the object of the relevant class).

8. For overloading binary operators, one must make use of a member function, which take one explicit argument. Please note that, those operators overloaded by means of a friend function, should take two explicit arguments.

9. When binary operator overloading is done through a member function, the left hand operand must be an object of the relevant class.

10. Note that binary arithmetic operators such as +,-,* and / should return a value explicitly without changing their own arguments.

    Examples of Unary operators −

    - The increment (++) and decrement (--) operators.

- The unary minus (-) operator.
- The logical not (!) operator.

Note: In most cases, unary operators operate on the object for which they were called and normally, this can be done using prefix such as !obj, -obj, and ++obj but sometime they can also used as postfix as well like obj++ or obj--.

- Let us take an example to understand how unary minus (-) operator can be overloaded for prefix as well as postfix usage.

Program:

```
#include <iostream>
using namespace std;
class ABC {
  private:
    int f;          // 0 to infinite for measurement in feet
    int in;         // 0 to 12 for measurement in inches

  public:
    // required constructors
    ABC() {
      f = 0;
      in = 0;
    }
    ABC(int feet, int inches) {
      f=feet;
      inches=i;
    }

    // method to display distance
    void displayABC() {
      cout << "F: " << f << " I:" << in <<endl;
```

```
    }

    // overloaded minus (-) operator
    ABC operator- () {
       f = -f;
       in = -in;
       return ABC(f, in);
    }
};
int main() {
  ABC A1(11, 10), A2(-5, 11);

  -A1;                  // apply negation
  A1.displayABC();    // display A1
  -A2;                  // apply negation
  A2.displayABC();    // display A2
  return 0;
}
```

After compilation of the above cod, it produces the following result

```
F: -11 I:-10
F: 5 I:-11
```

4.4 Binaryoperator Overloading

- The operator which require two operands to perform operator overloading function are called as binary operator overloading.
- Binary operator overloading can be overloaded by
  i. By using member function
  ii. By using friend function
- In the former case, the function takes single argument, whereas in the latter case it takes two arguments.
- Let us understand the concept much better by using the following program.
- One can use binary operators very often such as addition (+) operator, subtraction (-) operator and division (/) operator.
- Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

Program:

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

class ABC {
  double l;     // l=Length of a box
  double b;     // b=Breadth of a box
  double h;      //h= Height of a box

  public:
   double getVol (void) {
     return l * b * h;
  }

  void setL( double len ) {
     l = len;
  }

  void setB( double bre ) {
     b = bre;
```

```
  }

  void setH( double hei ) {
    h = hei;
  }

  // Overload + operator to add two ABC objects.
  ABC operator+(const ABC& b) {
    ABC abc;
    abc.l = this->l + b.l;
    abc.b = this->b + b.b;
    abc.h = this->h + b.h;
    return abc;
  }
};
// Main function for the program
int main() {
  ABC abc1;              // Declare Box1 of type Box
 ABC abc2;              // Declare Box2 of type Box
 ABC abc3;              // Declare Box3 of type Box
  double vol = 0.0;     // Store the volume of a box here

  // abc1 specification
  abc1.setL(6.0);
  abc1.setB(7.0);
  abc1.setH(5.0);

  // abc2 specification
  abc2.setL(6.0);
 abc2.setB(7.0);
  abc2.setH(5.0);

  // volume of abc 1
  vol = abc1.getVol();
  cout << "Volume of ABC1 : " << vol <<endl;

  // volume of abc 2
  vol = abc2.getVol();
  cout << "Volume of ABC2 : " << vol <<endl;
```

```
  // Add two object as follows:
  abc3 = abc1 + abc2;

  // volume of box 3
  vol = abc3.getVol();
  cout << "Volume of ABC3 : " << vol <<endl;
  return 0;
}
```

After executing the program, it produces the following result −

```
Volume of ABC1 : 210
Volume of ABC2 : 1560
Volume of ABC3 : 5400
```

- Similarly relational such as (<, >, <=, >=, ==, etc.) operator can be overloaded which can also be used to compare C++ built-in data types.
- Let us take an example which explains how a < operator can be overloaded. Similar logic can also be applied to overload other relational operators.

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

class DistDemo {
  private:
    int f;          // 0 to infinite
    int in;         // 0 to 12

  public:
    // required constructors
    DistDemo() {
      f = 0;
      in = 0;
    }
    DistDemo (int feet, int inch) {
      f = feet;
      in = inch;
    }

    // method to display distance
```

```
    void displayDist() {
      cout << "F: " << f << " I:" << in <<endl;
    }

    // overloaded minus (-) operator
    DistDemo operator- () {
      f = -f;
      in = -in;
      return DistDemo(f, in);
    }

    // overloaded < operator
    bool operator <(const DistDemo& d) {
      if(f < d.f) {
        return true;
      }
      if(f == d.f && in < d.in) {
        return true;
      }
      return false;
    }
};
int main() {
  DistDemo DD1(11, 10), DD2(5, 11);

  if( DD1 < DD2 ) {
    cout << "DD1 is less than DD2 " << endl;
  } else {
    cout << "DD2 is less than DD1 " << endl;
  }
  return 0;
}
```

On successful execution of the above code it produces the following result −

DD2 is less than DD1

4.5 Overloading Subscript Operator

- The Subscript Operator or Array Index Operator is denoted by '[]'.
- This operator is mainly used with arrays to retrieve and manipulate the elements of an array.
- This operator is generally of the type of binary or n-ary and is denoted as:
    1. postfix/primary expression
    2. expression
- The postfix expression, also known as the primary expression, is a pointer value such as array or identifiers and the second expression is an integral value.
- Enumerated values are included in the second expression
- **Syntax**:

    postfix-expression[expression];

Example:

RamLaxman[10];

Here the RamLaxman is an array and the above statement print the value which is held by RamLaxman at index position 10.

Note: The postfix expression followed by the subscript operator is the pointer and it can be an integral value but the one must keep in mind that one of expression among two expressions must be a pointer value and it does not matter whether the second one is of an integral order or not.

```
// CPP program to demonstrate []
// operator
#include <iostream>
using namespace std;
int main()
{
```

```
    char name[] = "RamLaxman Akshay Rajesh Suyesh";
    // Both of the statement prints same thing
    cout << name[5] << endl;
    cout << 5 [name] << endl;
    return 0;
}
```

Output:

```
a
a
```

OUTPUT

```
a
a
```

Explanation:

In the above example both "cout" statement provides similar output due to the exclusive property of the subscript operator. The compiler reads both the statement in a similar way, so there is no difference between the ***(name + 5)** and the ***(5 + name)**.

Positive and Negative subscripts

- In C++, array index starts with value 0 and the highest value which can be stored in an array [size – 1].
- However, we know that C++ supports both positive and negative subscripts.
- Please note that the Negative subscripts must fall within array boundaries; if they do not, the results are highly unpredictable.
- The following code shows positive and negative array subscripts:

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

// Driver Method
int main()
{
   int intArr[1024];
```

```
    for (int x = 0, y = 0; x< 1024; x++) {
        intArr[x] = y++;
    }

    // 512
    cout << intArr[512] << endl;

    // 257
    cout << 257 [intArr] << endl;

    // pointer to the middle of the array
    int* midArr = &intArr[512];

    // 256
    cout << midArr[-256] << endl;

    // unpredictable, may crash
    cout << intArr[-256] << endl;
}
```

Output:

```
512
257
256
0
```

Note: In the above program, the negative subscript in the last line can produce a run-time error since it points to an address position at -256 positions which can be lower in memory and violates the origin of the array. The pointer midArr is specifically initialized to the middle of intArr; to derive the use of both positive and negative array indices simultaneously. Array subscript errors fails to generate compile-time errors, but instead they might yield unpredictable results.

4.6 Type Conversion Operators

- In C++, type conversion is a technique which allows one to convert the data from one form to another.

- C++ has a rich collection of data types ranging from the basic (primitive) data types to the User Defined (Object) data types.
- In this section, we are going to learn about the conversion of these data types from one form to other.
- In C++, we can convert from one form to another as listed below
    1. Automatic conversion
    2. Primitive to Object
    3. Object to Primitive
    4. Object to Object

4.6.1 Automatic Conversion

- This is pretty straight forward process and is often done implicitly.
- For example,

int x1;

float x2 = 0.316;

In order to assign the value of float to x1, we can write

x1= x2;

Note: When the above code is compiled, the compiler will not generate any error(x1 is of integer type whereas x2 is of float type) and instead it will be handled implicitly by running some internal routine to convert the float value to integer.

- However the programmer can also give commands to the compiler by writing code to convert the float type to integer type. This type of conversion is known as ***explicit conversion*** of basic type data.
- Let us understand the concept much better by using the following program.

Program:

```
#include <iostream>
#include <string>
using namespace std;
void main(void) {
    int x1;
    float x2 = 5.825;
```

```
        x1=x2;
        cout<<x1;
}
```

Output:

Program analysis

In the above program, we have declared two variables, x1 of type integer and x2 of type float and we initialized float with value 5.825. As we have stored the value of float variable in integer variable by using equals sign operator. In this the compiler does run the conversion routine automatically and stock the integer type value of float into integer variable.

4.6.2 Primitive to Object

- Conversion from primitive to object data type can be best explained with the help of code.
- Consider a class called Dist which will have values in meters to distance class; where meter is a float type primitive type and distance is an object data class.
- We need to apply programming logic in order to convert the float to user defined class.
- The amazing thing is that we can perform something like this

  Dist = float;

- In the above statement we will be storing float values in distance class and will show it.

Program:

```
#include <iostream>
using namespace std;
const float MTF=3.280;
class Dist {
      int f;
      float in;
      public:
      Dist()  //Distance Constructor {
            f=0;
            in=0.0;
      }
      Dist(float nom)  //Single Parameter constructor {
            float fif= MTF * nom;
            f=int(fif);
            in=12*(fif-f);
      }
      void showdist()  // Method to display converted values {
            cout<<"Converted Value is: "<<f<<"\' feets and "<<in<<'\"'<<"
inches.";
      }
};
int main() {
      cout <<"Float to distance
conversion.\n*******************************\n";
      float m;
      cout<<"Enter values in meter:";
      cin >>m;
      Dist dist = m;
      dist.showdist();
}
```

Output:

Float to distance conversion

****

Enter values in meter: 7

Converted values is: 22.96 feet and 275.59 inches

Program analysis:

- In the above program, class Dist is created with two member variables: integer type ‘f’ and float type ‘in’.
- A no argument constructor is created which will initialize the values of f and in to 0 and 0.0 respectively.
- Next, it has a constructor that takes a float type variable as an argument.
- Within this constructor, we multiply the passed float type variable with 3.280833 which is stored in constant MTF variable.
- Next, we will multiply the passed float parameter with this number because passed variable will contain meters. And one meter contains 3.280833.
- In the above program,class Dist has distance expressed in feet and inches, therefore we converted the float to feet. Then we truncated the decimal part of the feet using

  f=int(fif)

4.6.3 Object to Primitive

- In C++, while doing conversion from Object type to Primitive type, a whole new concept is involved in this conversion called as overloading casting operator.
- In overloaded casting technique, the operator function actually overloads the built in casting operator.
- Syntax of overloaded casting operator is simple.

```
operator type()
{
.
.
.
}
```

- This kind of function will have no return type and without any arguments. The 'operator' is the keyword that has to be used followed by basic data type.
- Let us understand this by example. If one needs to overload the float operator can be done by operator float() {}.
- Consider the following condition needs to be satisfies in order to overload casting operator
    1. It does not have any return type.
    2. It does not allow to pass any parameters to an overloaded casting operator.
    3. Lastly, it should be defined inside a class definition. In this case, the class definition will be the object type that we want to convert into a primitive type whose casting operator needs to be overloaded.
- Let's understand how to implement overloaded casting operator by the following program.

```
#include <iostream>
using namespace std;
const float MTF=3.280833;
// Meter to feet
class Dist {
    int f;
    float in;
    public:
    Dist()        // Default Constructor {
        f=0;
        in=0.0;
    }
    Dist(int ft, float inc)  //two arguements constructor {
        f=ft;
        in=inc;
    }
```

```
    operator float()    //overloaded casting operator {
        float FIF=in/12;
        FIF+=float(f);
        return (FIF/MTF);
    }
};
int main() {
    int feet;
    float inches;
    cout <<"Enter distance in Feet and Inches.";
    cout<<"\nFeet:";
    cin>>feet;
    cout<<"Inches:";
    cin>>inches;
    Dist dist(feet, inches);
    float m=dist;
    // This will call overloaded casting operator
    cout<<"Converted Distance in Meters is: "<< m;
}
```

Output:

Enter the distance in Feet and Inches:

Feet: 22.96

Inches: 275.59

Converted Distance in Meters is: 7

Program analysis

- In the above program, we have an overloaded casting operator which overloads float type basic data type.
- The logic is applied inside this operator definition, to merge feet and inches to get a consolidate value in meters.

float FIF=in/12;

- Similarly, we applied a formula which is to be casted the feet member variable to float so that it can be added to FIF. Now FIF contain f + in.
- Next, we need to divide this value by MTF variable which contains value 3.280833.

4.6.4 Object to Object

- In this type of conversion, one can assign data that belongs to a particular class type to an object that belongs to another class type.
- Let us create two classes 'X' and 'Y'. In order to allocate the details that belong to class 'X' to an object of class 'Y' then this can be defined by –

Y(object of class Y) = X(object of class X)

where '=' has been overloaded for objects of class type 'Y'.

Program:

```
#include <bits/stdc++.h>
using namespace std;
class Demo {
   string x = "Hello World";
  public:
   string get_str ()
   {
      return (x);
   }
   void display()
   {
      cout << x << endl;
   }
};
 class Demo1 {
   string y;
  public:
```

```
    void operator =(Demo a)
  {
    y = a.get_str();
  }
   void display()
  {
    cout << y << endl;
  }
};
 int main()
{
  // Creating object of class Class_type_one
  Demo d;
  Demo d1;
  D1 = d;
  d.display();
  d1.display();
  return 0;
}
```

Output:

Hello World

Hello World

4.7 Advantages

- Operator overloading allow the C++ developers to use notation closer to the target domain. For example we can subtract two matrices by writing X1 - X2 rather than writing X1.subtract(X2).
- Operator overloading provides consistent syntactic support of right from built-in types till user-defined types.
- The basic goal is to make programs easier to understand.

- By overloading standard operators on a class, you can exploit the intuition of the users of that class. This lets users program in the language of the problem domain rather than in the language of the machine.
- The ultimate goal is to reduce both the learning curve and the defect rate.

4.8 Drawbacks

There are some limitations on operator overloading that are not very important for the practicing programmer, at least not at this stage.

The following operators cannot be overloaded

operator :: (scope),

operator .* (member object selector),

operator . (class object ...etc)

4.9 Summary

- In this unit, we have seen how the normal C++ operators can be given new meanings when applied to user-defined data types.
- Operator overloading can be implemented by
  - Using member function
  - Non-Member function
  - Friend function
- The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.
- Closely related to operator overloading is the issue of type conversion. Some conversions take place between user defined types and basic types.

4.10 Unit End Exercises

1. Overload the addition operator (+) to assign binary addition. The following operation should be supported by +.
   110010 + 011101 = 1001111

2. What will be the output of the following program snippet? Explain.

   ```
   int x;
   float y = 11.1883;
   ```

```
x=y;
cout<<x;
cout<<y;
```

3. Which operators are not allowed to be overloaded?

4. What are the differences between overloading a unary operator and that of a binary operator? Illustrate with suitable examples.

5. Why is it necessary to convert one data type to another? Illustrate with suitable examples.

6. How many arguments are required in the definition of an overloaded unary operator?

7. When used in prefix form, what does the overloaded + + operator do differently from what it does in postfix form?

8. Write the complete definition of an overloaded + + operator that works with the string class from the STRPLUS example and has the effect of changing its operand to uppercase. You can use the library function toupper ( ), which takes as its only argument the character to be changed, and returns the changed character.

9. Write a note on unary operators.

10. What are the various rules for overloading operators?

Test Your Knowledge

1. What is a binary operator?

   a) Operator that performs its action on a single operand

   b) Operator that performs its action on two operand

   c) Operator that performs its action on three operand

   d) Operator that performs its action on any number of operands

2. Which is the correct example of a binary operator?

   a) ++

   b) —

   c) Dereferencing operator(*)

   d) +

3. Which is the correct example of a unary operator?

   a) &

   b) ==

   c) —

   d) /

4. Which is called ternary operator?

   a) ?:

   b) &&

   c) |||

   d) ===

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

```
#include <iostream>
#include <string>
using namespace std;
class complex
{
    int i;
    int j;
     public:
    complex(int a, int b)
     {
          i = a;
          j = b;
    }
    complex operator+(complex c)
     {
          complex temp;
          temp.i = this->i + c.i;
```

```
            temp.j = this->j + c.j;
            return temp;
      }
      void show(){
            cout<<"Complex Number: "<<i<<" + i"<<j<<endl;
      }
};
 int main(int argc, char const *argv[])
{
      complex c1(1,2);
      complex c2(3,4);
      complex c3 = c1 + c2;
      c3.show();
      return 0;
}
```

a) 4 + i6

b) 2 + i2

c) Error

d) Segmentation fault

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

```
#include <iostream>
#include <string>
using namespace std;
class complex
{
      int i;
      int j;
       public:
      complex(){}
      complex(int a, int b)
```

```
    {
        i = a;
        j = b;
    }
    complex operator+(complex c)
    {
        complex temp;
        temp.i = this->i + c.i;
        temp.j = this->j + c.j;
        return temp;
    }
    void show(){
        cout<<"Complex Number: "<<i<<" + i"<<j<<endl;
    }
};
 int main(int argc, char const *argv[])
{
    complex c1(1,2);
    complex c2(3,4);
    complex c3 = c1 + c2;
    c3.show();
    return 0;
}
```

a) Complex Number: 4 + i6

b) Complex Number: 2 + i2

c) Error

d) Segmentation fault

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

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

```
    int i;
    int j;
   public:
    complex(){}
    complex(int a, int b)
     {
          i = a;
          j = b;
    }
    complex operator+(complex c)
     {
          complex temp;
          temp.i = this->i + c.i;
          temp.j = this->j + c.j;
          return temp;
    }
    void operator+(complex c)
     {
          complex temp;
          temp.i = this->i + c.i;
          temp.j = this->j + c.j;
          temp.show_poss();
    }

    void show(){
          cout<<"Complex Number: "<<i<<" + i"<<j<<endl;
    }

    void show_poss(){
          cout<<"Your result after addition will be: "<<i<<" + i"<<j<<endl;
    }
};
```

```
 int main(int argc, char const *argv[])
{
    complex c1(1,2);
    complex c2(3,4);
    c1 + c2;
    return 0;
}
```

a) Complex Number: 4 + i6

b) Complex Number: 2 + i2

c) Error

d) Segmentation fault

8. Which operator should be overloaded in the following code to make the program error free?

```
#include <iostream>
#include <string>
using namespace std;
class Box{
    int capacity;
public:
    Box(){}
    Box(double capacity){
        this->capacity = capacity;
    }
};
int main(int argc, char const *argv[])
{
    Box b1(10);
    Box b2 = Box(14);
    if(b1 == b2){
        cout<<"Equal";
    }
```

```
    else{
        cout<<"Not Equal";
    }
    return 0;
}
a) +
b) ==
c) =
d) ()
```

9. Give the function prototype of the operator function which we need to define in this program so that the program has no errors.

```
#include <iostream>
#include <string>
using namespace std;
class Box{
    int capacity;
public:
    Box(){}
    Box(double capacity){
        this->capacity = capacity;
    }
};
 int main(int argc, char const *argv[])
{
    Box b1(10);
    Box b2 = Box(14);
    if(b1 == b2){
        cout<<"Equal";
    }
    else{
        cout<<"Not Equal";
    }
    return 0;
}
a) bool operator==();
b) bool operator==(Box b){}
```

c) bool operator==(Box b);
d) Box operator==();
10. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class Box{
        int capacity;
public:
        Box(){}
        Box(double capacity){
                this->capacity = capacity;
        }
        bool operator<(Box b){
                return b.capacity < this->capacity? true : false;
        }
};

int main(int argc, char const *argv[])
{
        Box b1(10);
        Box b2 = Box(14);
        if(b1 < b2){
                cout<<"B1's capacity is small";
        }
        else{
                cout<<"B2's capacity is small";
        }
        return 0;
}
```

a) B1's capacity is small
b) B2's capacity is small
c) Error
d) Segmentation fault

4.11 Further Readings

Books

1. E Balagurusamy; *Object-Oriented Programming with C++*; Tata Mc Graw-Hill.
2. Herbert Schildt; *The Complete Reference C++*; Tata McGraw Hill.
3. Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.
4. Object Oriented Programming using C++-Lovely Professional University notes

Online links

1. http://www.mochima.com/tutorials/strings.html
2. http://www.exforsys.com/tutorials/c-plus-plus/operator-overloading-partii.html
3. https://www.geeksforgeeks.org/c-operator-overloading-question-1/
4. https://www.sanfoundry.com/cplusplus-programming-questions-answers-operator-overloading-2/
5. https://www.geeksforgeeks.org/overloading-subscript-or-array-index-operator-in-c/

❖❖❖

UNIT 3

5

POINTERS IN C++

Unit Structure

5.0 Objective

At the end of this unit, students will be able to:

- Have good understanding regarding pointers.
- Determine how to access address of a given variable.
- Understand the how to implement explicit and mutable pointers
- Understand the differences between pointer to an array and array of pointers.
- Understand the similarity and differences between pointer to a constant and constant pointer.
- Implement various types of pointers in C++.
- Identify the understanding pointers
- Recognize the declaring and initializing pointers
- Describe the pointer to pointer
- Explain the pointer to a function
- Discuss the dynamic memory management

5.1 Explicit and Mutable Pointers

Pointers

- Pointer is a variable in C++ which points to the address of another variable.
- They have data type just like variables, for example an integer type pointer can hold the address of an integer variable and an character type pointer can hold the address of char variable.

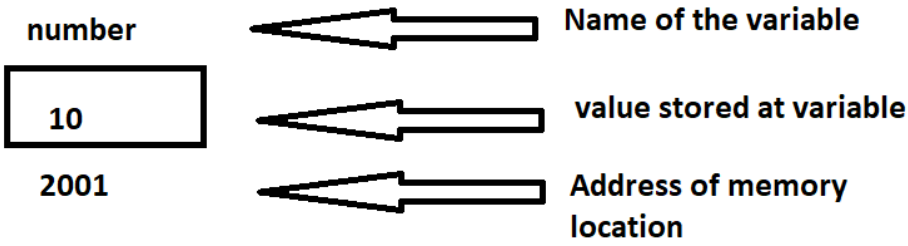


Figure 5.1 Declaration of a variable

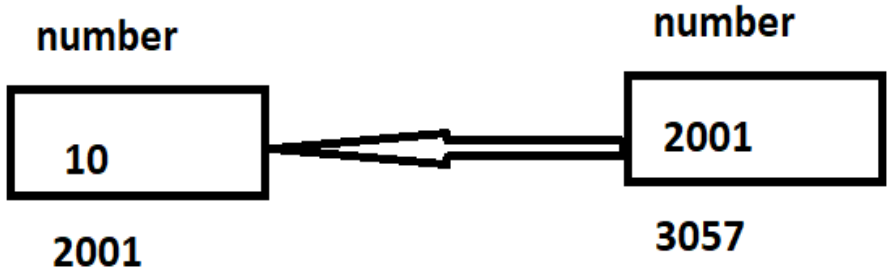


Figure 5.2 Declaration of a pointer variable

- Let "p" be the pointer variable which holds the address of variable num.
- Thus, we can access the value of "number" by the pointer variable 'p'. Thus, we can say "p points to number".
- Diagrammatically, Fig 5.2 shows the declaration of a pointer variable.

5.2 Pointer and Address of Operator

- In C++, we can acquire an address of a variable by adding a symbol of ampersand sign (&) prior to the name of the variable. This is known as address of operator.

For example,
```
x = &a;
```

- In the above example, the statement would assign the address of variable "a" to x; by adding a symbol (&) prior to the name of the variable "a" with the *address-of operator* (&), we are no longer assigning the content of the variable itself to x, but its address.
- Note that the actual address of a variable in memory will be known only at the execution of the code.
- To simplify the things, let us assume the variable "a" is placed during runtime in the memory address 1776.
- Let us consider the following code fragment:

```
1  a = 25;
2
3  x = &a;

   y = a;
```

- The diagram below depicts the values assigned in each of the variable after execution of the code.

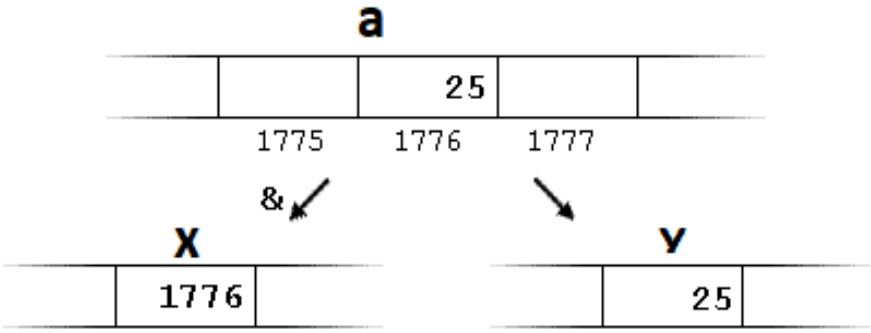


Figure 5.3 Values assigned to each variable

- First, we have assigned the value 25 to a (a variable whose address in memory we assumed to be 1776).
- The second statement assigns x to the address of a, which we have assumed to be 1776.
- Finally, the third statement, assigns the value contained in a to y. This operation is repeated several times in preceding chapters.
- One striking difference between the second and third statements are the position of address-of operator(&)
- Note that in C++, if we have a variable which will store the address of another variable (like x in the previous example) we call such variables as a *pointer*.
- Pointers are a very influential feature of any programming language that has many uses in lower level programming. A bit later, we will see many different types of pointers.

5.3 Pointer to an Array

- Let us understand this concept by the following example.

Syntax:

data_type (*var_name)[size_of_array];
where var_name-> name of the pointer variable

size_of_array-> maximum size of an array

Consider the following code

```
#include<iostream.h>
int main()
{
  int a[5] = { 10, 20, 30, 40, 50 };
  int *ptr = a;
  cout<< ptr;
  return 0;
}
```

Output:

0x7ffeef272b70

- In the above program, pointer variable(ptr) which points to the 0^{th} element of the array. Instead of declaring a pointer which points to only one element, we can declare a pointer which can point to whole array This is called as **array of pointer.**

  Example:

  int (*ptr)[5];

- In the above example, we have pointer variable(*ptr)* is pointer that points to an array of 5 integers. Please note that subscript have higher precedence than indirection, therefore, they are enclosed in parentheses it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 5 integers'.

- Let us understand in more detail with the following code.

Program:

```
#include <iostream>
using namespace std;
int main()
{
    // Pointer to an integer
```

```
    int *x;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int a[5];

    // Points to 0th element of the arr.
    x = a;

    // Points to the whole array arr.
    ptr = &a;
    cout << "x =" << x <<", ptr = "<< ptr<< endl;
    x++;
    ptr++;
    cout << "x =" << x <<", ptr = "<< ptr<< endl;
    return 0;
}
```

- x: is pointer to 0th element of the array a, while ptr is a pointer that points to the whole array a.
- The base type of x is int while base type of ptr is 'an array of 5 integers'.
- Please note that if we write ptr++, then the pointer ptr will be shifted ahead by 20 bytes.
- Let us understand by following diagram

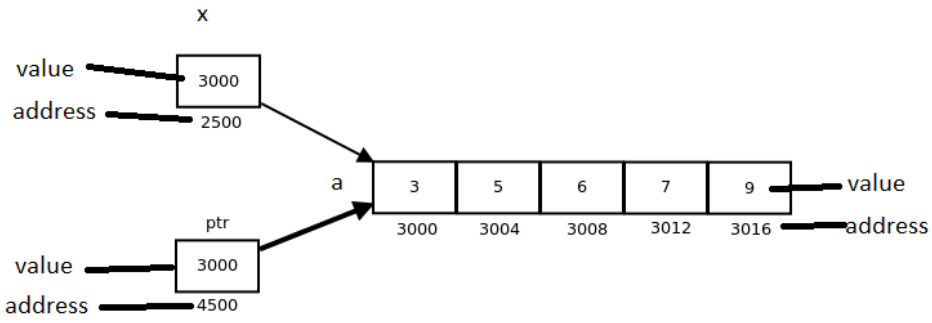


Figure 5.4: Working of array of pointers

In the above figure, whenever we dereferenced a pointer expression we get a value pointed to by that pointer expression. Please note that whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

5.4 Array of Pointers

- When we have an array which consists of variables of pointer type, we have an variable which is a pointer which address to some other element. This concept is called as **array of pointers**.
- Let us take an example and create an array of pointer holding 10 integer pointers; then its declaration would look like:

  int *ptr[10];
- In the above statement, we have created an array of pointer named as ptr, and it allocates 10 integer pointers in memory.
- The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's us consider the following example

  int x; // variable declaration.

  ptr[1] = &x;
- In the above code, we are assigning the address of 'x' variable to the second element of an array 'ptr'.
- Similarly, by using the concept of derefencing, we can derive the value of pointer variable

  1. *ptr[1];

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
   int ptr1[15]; // integer array declaration
   int *ptr2[15]; // integer array of pointer declaration
   std::cout << "Enter fifteen numbers :" << std::endl;
```

```
for(int i=0;i<15;i++)
{
   std::cin >> ptr1[i];
}
for(int i=0;i<15;i++)
{
   ptr2[i]=&ptr1[i];
}
// printing the values of ptr1 array
std::cout << "The values are" << std::endl;
for(int i=0;i<15;i++)
{
   std::cout << *ptr2[i] << std::endl;
}
}
```

- In the above program, we declare an two arrays. i.e; one of integer type and second of integer pointers. We have used the 'for' loop, which iterates through the elements which are the part of an array 'ptr1'.
- Please note on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

5.5 Pointer Arithmetic

- Pointer arithmetic can be applied by using arithmetic operations on pointer variable.
- This can be achieved by using following operators on pointers: **++, --, +, and –**
- Let us consider, we have a ptr which is of integer type pointer which points to the address 5000.
- If we now perform ptr++, the pointer will now point to location 5004(4 bytes of memory reserved for integers). The best part of this is that though it allows to shift to new location in array, it absolutely makes no changes to the value stored at a given location.

- If instead of integer pointer, if we use character pointer, then it will point to next character location within an array if we use it by ptr++.
- Instead of integer point, if we use character pointer which is pointing to the base address at 1000, then above operation will point to a new location 1001 because next character will be available at 1001(character is 1 byte in size).
- The two most popular methods of pointer arithmetic are
    1. Incrementing a Pointer
    2. Decrementing a Pointer

1. Incrementing a Pointer

- Let us understand this in more detail with the following program

```
#include <iostream>
using namespace std;
const int MAX = 5;
int main () {
  int  v[MAX] = {50, 100, 150, 200, 250};
  int  *ptr;
  // let us have array address in pointer.
  ptr = v;
  for (int i = 0; i < MAX; i++) {
    cout << "Address =”<<ptr << endl;
    cout << "Value = "<<i<<endl;
    cout << *ptr << endl;
    ptr++; //increments to new location
  }

  return 0;
}
```

On successful compilation and execution of the above code, it produces output as follows −

```
Address= 0xbfa088b0
Value = 50
Address = 0xbfa088b4
Value = 100
Address = 0xbfa088b8
Value = 150
Address = 0xbfa088b12
Value = 200
Address = 0xbfa088b16
Value = 250
```

2. Decrementing a Pointer

- In this case, similar contemplations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as discussed below

Program:

```
#include <iostream>
using namespace std;
const int MAX = 5;
int main () {
  int  v[MAX] = {50, 100, 150, 200, 250};
  int  *ptr;
  // let us have address of the last element in pointer.
  ptr = &v[MAX-1];

  for (int i = MAX; i > 0; i--) {
    cout << "Address = ";
    cout << ptr << endl;
    cout << "Value = ";
    cout << *ptr << endl;
```

```
      // point to the previous location
      ptr--;
   }
   return 0;
}
```

On successful compilation and execution of the above code, it produces output as follows −

```
Address= 0xbfa088b16
Value = 250
Address = 0xbfa088b12
Value = 200
Address = 0xbfa088b8
Value = 150
Address = 0xbfa088b4
Value = 100
Address = 0xbfa088b0
Value = 50
```

5.6 Pointer to A Constant

- In this case, a non-const pointer points to a constant value.
- We use const keyword before declaring the datatype and variable name. Let us see the example.

```
1  const int v = 15;
2  const int *ptr = &v; // ptr is a non-const pointer
3  *ptr = 16; // impossible statement, cnt change value of constant
```

- In the above statement, ptr points to a integer type constant.

Let us take another good example:

```
1 int v = 15; // v is not constant
2 const int *ptr = &v; // allowed
```

- The above statement is a very good example of how a pointer to a constant variable can point to a non-constant variable.
- Now let us take one more example:

```
1 int v = 5;
2 const int *ptr = &v;
3 v = 6;
```

But the following is not allowed and is considered as illegal statement:

```
1 int v = 5;
2 const int *ptr = &v;
3 *ptr = 6; //
```

Note: The above statements are invalid as a pointer to a const value is not const itself, thus enabling the pointer to be redirected to point at other values.

5.7 Constant Pointer

- A mechanism where we make pointer itself a constant is called as constant pointer.
- We know that a const pointer is a pointer whose value are fixed after initialization
- We make use of const keyword inorder to declare a const pointer as shown below

```
int v = 15;
int *const ptr = &v;
```

- We are also aware that a const pointer must be initialized to a some known value upon declaration. Please keep in mind that a const pointer will always point to the same address.
- In the above case, ptr will always point to the address of value until it is destroyed or goes out of scope.

```
int v1 = 15;
int v2 =16;
int * const ptr = &v1; //allowed
ptr = &v2; //not allowed
```

- One can change the value being pointed by implementing dereferencing the const pointer as given below

```
int v = 15;
int *const ptr = &v; //allowed
*ptr = 16; //allowed
```

5.8 Pointer Declaration & Initialization

- Now let us begin with how to declare and initialize a pointer variable. The general form of a pointer variable is given by
- **Syntax:**

```
datatype *pointer_name;
```

- Please note that the Data type of a pointer and the data type of the variable to which the pointer variable is pointing should be the same and cross data types are not allowed here.
- The exception to this is void type pointer which works with all data types, but is rarely used.

Here are a few good examples on how to declare a pointer variables:

```
int *a     // pointer to integer variable
float *b;      // pointer to float variable
double *c;     // pointer to double variable
char *d;       // pointer to char variable
```

Initialization of C Pointer variable

- The process of assigning an address of a variable to a pointer variable is called as **pointer initialization**.
- They only contain address of a variable of the same data type.
- In C++ language, we use **address operator (&)** to determine the address of a variable.
- By using & operator (immediately prefixing a variable name) we can fetch the address of the variable associated with it.

```
#include<iostream.h>
void main()
{
   int x = 20;
   int *ptr;      //declaring a pointer
   ptr = &x;      //initializing a pointer
}
```

Please note that pointer variables always point to variables of same datatype. Consider the following example:

```
#include<iostream.h>
void main()
{
   float x;
   int *ptr;
   ptr = &x;      // ERROR, type mismatch
}
```

Please remember to assign a NULL value to your pointer variable If you are not sure about which variable's address to assign to a pointer variable while performing declaration of variables. The pointer which has been assigned with a NULL value is called as a **NULL pointer**. Consider the example of NULL pointer listed below

```
#include <iostream.h>
int main()
{
   int *ptr = NULL;
   return 0;
}
```

Using the pointer or Dereferencing of Pointer

- Once a address of a variable is assigned to a pointer, in order to access the value of the variable, pointer is dereferenced using the indirection operator or deferencing operator(*).
- Consider the following example

```
#include <iostream.h>
int main()
{
   int x, *ptr;  // declaring the variable and pointer
   x = 10;
   ptr = &x;     // initializing the pointer
   cout<< *ptr;    //this will print the value of 'x'
   cout<< *&x;   //this will also print the value of 'x'
   cout<< &x;    //this will print the address of 'x'
   cout<< ptr;     //this will also print the address of 'x'
   cout<< &ptr;    //this will print the address of 'ptr'
   return 0;
}
```

Points to remember while using pointers

1. Prefix asterisk (*) in variable declaration indicates that the variable is a pointer.
2. If we precede the variable name with Ampersand (&) keyword would help us to fetch the address of given variable.
3. Unlike normal variable, there is a special case where a pointer variable will store only stores the address of a variable.
4. In order to access the value of a certain address stored by a pointer variable, we use (*) which can be read as ‘value at’.
5. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as **'value at'**.

5.9 Types of Pointers

5.9.1 VOID POINTER

- It is the only pointer which lacks the data type associated with it.
- They are basically used to hold address of any type and can be converted to any another type.

```
int x = 20;
char y = 'a';
void *ptr = &x;              //  address of int 'x' -> ptr
ptr = &y;                    // address of char 'y' ->ptr
```

Advantages of void pointers:

1) Please note that malloc() and calloc() always return void * type data and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)
{
   int *a = (int *) malloc(sizeof(int) * p);
}
```

Points to remember

1. In the above code, we have explicitly typecast return value of malloc to (int *) as in C++ it is mandatory requirement. This may note be done while compiling in C language.
2. No dereferencing is possible in case of void pointers.

5.9.2 NULL POINTER

- They are basically used in the special case where we are not sure about the exact address to be assigned to a pointer variable.
- Assignment of NULL pointer is done at the time of variable declaration.
- They are basically assigned by using NULL keyword.

- The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream.
- Consider the following demonstration−

```
#include <iostream>
using namespace std;
int main () {
  int  *p = NULL;
  cout << "Value of Pointer= " << p;
  return 0;
}
```

- On successful compilation and execution of the above mention code, it produces the following outcome −

```
Value of Pointer = 0
```

- Please note that modern day operating systems do not permit programs to access memory at address 0 as it is reserved by operating system.
- If a pointer is pointing to memory address 0, it signifies that the pointer is not interested to point to accessible memory location.
- It is a practice to make use an if statement in order to check for the null pointer which is describe as follows −

```
if(ptr)
if(!ptr)
```

5.9.3 DANGLING POINTER

- It is a pointer which is pointing to memory location which is freed recently or deleted is called as dangling pointer.
- This is done by using three different ways given below

1. **De-allocation of memory**

- Let us understand this by the following code

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

```
int main()
{
   int *ptr = (int *)malloc(sizeof(int));
   free(ptr);  //pointer becoming a dangling pointer
   // No more a dangling pointer
   ptr = NULL;
}
```

2. **Function Call**

```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
#include<iostream.h>
int *fun()
{
   int x1 = 15;
   return &x1;
}
// Driver Code
int main()
{
   int *p1 = fun();
   fflush(stdin);
cout<<*p;
   return 0;
}
```

Output:

A garbage Address

The above problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.

```
// The pointer pointing to local variable doesn't
// become dangling when local variable is static.
#include<stdio.h>
int *fun()
{
```

```
    // x now has scope throughout the program
    static int x = 5;
    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // Not a dangling pointer as it points
    // to static variable.
    printf("%d",*p);
}
```

Output:

```
 5
```

3. **Variable goes out of scope**

```
4.  void main()
5.  {
6.      int *ptr;
7.      .....
8.      .....
9.      {
10.         int ch;
11.         ptr = &ch;
12.     }
13.     .....
14.     // Here ptr is dangling pointer
}
```

5.10 Dynamic Memory Allocation

- In C++, dynamic memory allocation and deallocation is done by using **new** and **delete** operators.
- In C++, dynamically allocated memory is allocated by Heap.
- Dynamic memory allocation provides the flexibility to the programmers to increase and decrease memory as per requirement. Examples of such dynamic allocation are linked list, tree etc.
- In previous chapter, let us recall the following statement

  int *x=new int[20]

Note: The main responsibility of the programmer here is to deallocate the memory which is no longer used and hence this is done by using delete operator.

New Operator

- In C++, dynamic memory allocation is done by using **new** operator.
- When this operator is used, new operator initializes the memory and returns the address and initialized memory to the pointer variable.

Syntax:

pointer-variable = new data-type;

where, pointer-variable is the pointer of type data-type.

Examples:

1. int *p = new int;
2. int *p = new int(25);
3. float *q = new float(75.25);

Delete Operator

- In C++, programmer deallocate the memory dynamically allocated memory earlier by using **delete** operator.

Syntax:

delete pointer-variable;
where, pointer-variable -> pointer that points to the data object created by *new*.

Examples:

delete a;

delete b;

5.11 Advantages and Applications of Pointers

1. **Passing arguments by reference**. In this it serves two purposes

   (i) **To modify variable of function in other.** Example to swap two variables.

   (ii) **For efficiency purpose**. When passing huge structure without reference would create a copy of the structure (hence wastage of space).

2. **To access array elements.** This is done by using compiler internally uses pointers to access array elements.

3. **Returning multiple values.** Example returning cube and cube root of numbers.

4. **Dynamic memory allocation:** This is done by using pointers to dynamically allocate memory.

5. **Example linked list, tree, etc**. C++ references cannot be used to implement these data structures because references are fixed to a particular location.

6. **Performing system level programming where memory addresses are useful**. This is done by using shared memory used by multiple threads.

5.12 Summary

- A variable which hold the memory address of the location of another variable in memory is called as pointer. It can be defined as type * var_name ;

where type is a predefined C++ data type and var_name is the name of the pointer variable.

- The operator &, should be placed before a variable, also returns the memory address of its operand.

- The operator * when used returns the memory address of its operand.

- The operator * returns the data value stored in the area being pointed to by the pointer following it.

  - The pointer variables must always point to the correct type of data. Pointers must be initialized properly because uninitialized pointers result in the system crash.

  - In pointer arithmetic, all pointers increase and decrease by the length of the data type point to.

  - An array name is a pointer that stores the address of its first element. If the array name is incremented, It actually points to the next element of the array.

  - Array of pointers makes more efficient use of available memory. Generally, it consumes lesser bytes than an equivalent multi-dimensional array.

  - Functions can be invoked by passing the values of arguments or references to arguments or pointers to arguments.

  - When references or pointers are passed to a function, the function works with the original copy of the variable. A function may return a reference or a pointer also.

5.13 Unit End Exercises

1. Explain does pointer variable differ from simple variable?

2. How do we create and use an array of pointer-to-member-function?

3. How can we avoid syntax errors when creating pointers to members?

4. How can we avoid syntax errors when calling a member function using a pointer-tomember-function?

5. How do we pass a pointer-to-member-function to a signal handler, X event callback, system call that starts a thread/task, etc?

6. Find the syntax error (s), if any, in the following program:

```
{
int x [5], *y [5]
for (i = 0; i< 5; i++)
```

```
{ x [i] = I;
x[i] = i + 3;
y = z;
x = y;
}
```

7. Discuss two different ways of accessing array elements.
8. Write a program to traverse an array using pointer.
9. Write a program to compare two strings using pointer.
10. Write short note on dynamic allocation by using pointers.

5.14 Further Readings

Books

1. E Balagurusamy; *Object-Oriented Programming with C++*; Tata Mc Graw-Hill.
2. Herbert Schildt; *The Complete Reference C++*; Tata McGraw Hill.
3. Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.
4. Object Oriented Programming using C++-Lovely Professional University notes

Online links

1. http://www.mochima.com/tutorials/strings.html
2. http://www.exforsys.com/tutorials/c-plus-plus/operator-overloading-partii.html
3. http://publib.boulder.ibm.com/infocenter/comphelp/v8v101 index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr035.htm
4. http://www.cplusplus.com/doc/tutorial/pointers/

❖❖❖

UNIT 4

6

INHERITANCE AND POLYMORPHISM

Unit Structure

6.0 Objectives

The objective of the chapter is as follow

- To get familiar with concept of inheritance
- To understand different types of inheritance
- To understand the concept of member hiding, constructor inheritance

6.1 Introduction

Object Oriented Programming provides with an important characteristics known as inheritance which leads to the reusability of code and reducing the work of writing the same piece of code again

Inheritance are of different types which reflects different effect on its member functions depending on its visibility mode

6.2 Inheritance concept

- Inheritance is one of the important features supported by Object Oriented Programming.
- It defines the hierarchical relationship between classes and supports the concepts of division of code
- It supports creating a derived class using the attributes and the methods of based class
- Inheritance provides the reusability features which allows the code to be reusable. This mechanism will allow the same class to be used by adding some additional features to it.
- Resuability

Eg the base class is vehicle which is inherited by the train, car and flight

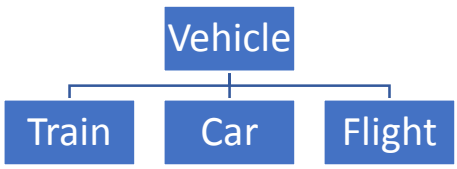


Figure 1. inheritance

6.3 Derivation of inheritance

Derived class:The class that inherits properties from another class is called Derived Class.

Base class: The class whose properties are inherited by derived class is called Base Class.

```
#include <iostream>
class base_classname
{
.... ... ....
};
class derived_classname : access_modifiersbase_classname
{
.... ... ....
};
```

A derived class is derived from a base class with different access control or modifiers.

These modifiers are as follow : public inheritance, protected inheritance or private inheritance.

6.3.1 Public mode :

If the based class is inherited in public mode then

public members of the base class will become public in derived class.

protected members will become protected in derived class

private members will remain not be accessed

```
#include <iostream.h>
#include <conio.h>
class base
{
    int a;
    protected:
```

```
        int b;
        public:
        int c;
        void get()
        {
        cout<<"enter 3 nos:";
        cin>>a>>b>>c;
        }
        int get_pri()
        {
  return a;
        }
};
class derived : public base
{
        public :
        void show()
        {
        cout<<get_pri()<<endl;
        cout<<b<<endl;
        cout<<c<<endl;
        }
};
void main()
{
clrscr();
      derived o;
o.get();
o.show();
getch();
}
```

6.3.2 Protected mode:

If the based class is inherited in protected mode then

public members of the base class will become protected in derived class.

protected members will become protected in derived class

private members will not be accessed in the derived class.

```
#include <iostream.h>
#include <conio.h>
class base
{
  int a;
protected:
int b;
public:
int c;
void get()
{
cout<<"enter 3 nos:";
cin>>a>>b>>c;
}
int get_pri()
{
  return a;
}
};
class derived : protected base
{
public :
void show()
{
```

```
 base::get();
cout<<get_pri()<<endl;
cout<<b<<endl;
cout<<c<<endl;
}
};
void main()
{
clrscr();
     derived o;
o.show();
getch();
}
```

6.3.3 Private mode:

It is the default mode of access in C++

If the based class is inherited in private mode then

public members of the base class will become private members in derived class.

protected members will become private in derived class

private members will not be accessed in derived class

```
#include <iostream.h>
#include <conio.h>
//inheritance
/*
class derivedclass : access modifier baseclass
{
      members of dc
};
*/
```

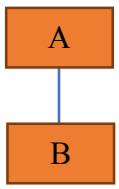
```
class base
{
        int a;


        protected:
        int b;
        public:
        int c;
        void get()
        {
        cout<<"enter 3 nos:";
        cin>>a>>b>>c;
        }
        int get_pri()
        {
          return a;
        }
};
class derived : private base
{
        public :
        void show()
        {
         base::get();
        cout<<get_pri()<<endl;
        cout<<b<<endl;
        cout<<c<<endl;
        }
};
```

```
void main()
{
clrscr();
     derived o;
o.show();
getch();
}
```

6.4 Types of inheritance

6.4.1 Single Inheritance:when a single derived class is inherited from the single based class

```
Syntax :
class derived_classname : access_modifierbase_classname
{
  //body of derived class
};
```



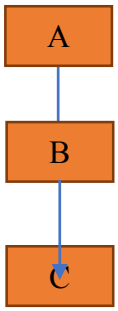

```
class Animals
{
   public:
 void commutes()
  {
cout<<"yes"<<endl;
   }
 };
class Horse: public Animal
```

```
  {
     public:
    void numberoflegs()
{
cout<<4;
    }
  };
int main(void) {
   Dog d1;
   d1.commutes();
   d1.numberoflegs();
   return 0;
}
```

6.4.2 Multilevel inheritance: inherits derived class from another derived class

class derived_classname : access_modifier base_classname1, access_modifierbase_classname, ...

```
{
 //body
};
```



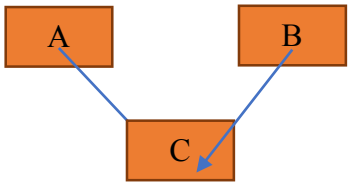

```
include <iostream>
 class Animal
{
  public:
```

```
 void commutes() {
cout<<"yes."<<endl;
 }
  };
  class Horse: public Animal
  {
     public:
void numberoflegs(){
cout<<"4"<<endl;
    }
  };
  class foal: public Horse
  {
     public:
    void tail() {
cout<<"yes";
    }
  };
int main(void) {
foal d1;
   d1.commutes();
   d1.numberoflegs();
    d1.tail();
    return 0;
}
```

6.4.3 Multiple inheritance : inheritance in which the derived class inherits from more then one base class

Syntax : class derived_classname : access_modifier base_classname1, access_modifierbase_classname, ...

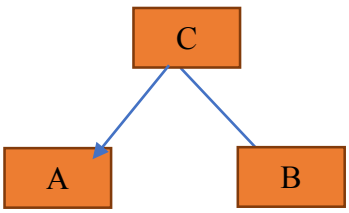
```
{
  //body
};
```




```
class individual
{
        int age;
        char name[10];
        public:
        int c;
        void get()
        {
        cout<<"enter name & age:";
cin>>name>>age;
              }
             void show()
            {
cout<<"name="<<name<<", age="<<age;
            }
};
class student
{
int marks;
public :
void get()
{
        cout<<"enter marks";
        cin>>marks;
}
```

```
void showmarks()
{
      cout<<" marks="<<marks;
}
};
class fulltime-student : public individual,public student
{
};
void main()
{
clrscr();
     fulltime o;
o.individual::get();//ambiguity resolved
o.student::get();//ambiguity resolved
o.show();
o.showmarks();
getch();
}
```

6.4.4 Hierarchical inheritance: inheritance in which more than one derived class inherits from the single base class

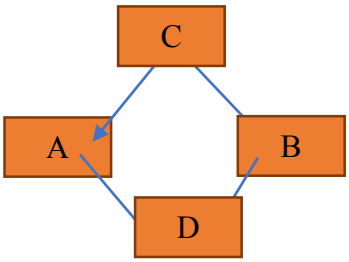



```
#include <iostream>
using namespace std;
class Shape
{
   public:
   int l;
```

```
    int b;
    void getdata(int a,int b)
    {
        l= a;
        b = b;
    }
};
class Rectangle : public Shape
{
    public:
    int area_rect()
    {
        int area = l*b;
        return area;
    }
};
class Triangle : public Shape
{
    public:
    int area_triangle()
    {
        float area = 0.5*l*b;
        return area;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
```

```
    int length,breadth,base,height;
cout<< "Enter the length and breadth of a rectangle: " ;
cin>>length>>breadth;
r.getdata(length,breadth);
    int result_rect = r.area_rect();
cout<< "Area of the rectangle is : ";
cout<< "Enter the base and height of the triangle: " ;
cin>>base>>height;
t.getdata(base,height);
    float result_tri = t.area_triangle();
cout<<"Area of the triangle is : " ;
    return 0;
}
```

6.4.5 Multipath inheritance: Inheritance in which a derived class is formed by two base classes and these two base classes have one common base class is called multipath inheritance




```
class emp
{
      int age;
      char name[10];
      public:
      int c;
      void get()
      {
      cout<<"enter name & age:";
      cin>>name>>age;
      }
```

```
    void show()
    {
    cout<<"name="<<name<<" age="<<age;
    }
};
class fulltime :  public emp
{
    int sal;
    public :
    void getsal()
    {
    cout<<"enter sal";
    cin>>sal;
    }
    void showsal()
    {
    cout<<" sal="<<sal;
    }
};
class contract : public emp
{
    int workinghrs;
    int wagesper_hr;
    public:
    void gethr_wage()
    {
    cout<<"enter working hrs & wages per hr";
    cin>>workinghrs>>wagesper_hr;
    }
    void showcal_sal()
    {
    cout<<" Total salary="<<workinghrs*wagesper_hr;
    }
};
class performance:publicfulltime,public contract
{
    int score;
```

```
      public:
      void getscore()
      {
      cout<<"enter score";
      cin>>score;
      }
      void showscore()
      {
      cout<<" rating="<<score;
      }
};
void main()
{
clrscr();
cout<<"\nEnter Fulltime emp data"<<endl;
     fulltime f;
f.get();
f.getsal();
performance p;
p.getscore();
f.show();
f.showsal();
p.showscore();
cout<<"\nEnter Contract emp data"<<endl;
     performance p1;
    p1.contract::get();
     p1.getsal();
     p1.getscore();
     p1.contract::show();
     p1.showsal();
     p1.showscore();
getch();
}
```

6.5 Function hiding

- In Method Hiding takes place when the derived class has function with same name as that of the function in base class, then the derived class function will hide all base class function with same name even if the signatures are different.
- The difference in return type or argument will not be considered in method hiding
- Method hiding gives a different implementation to the base class method with the change in signature
- To avoid the base class method form getting hidden, the scope resolution operator is used along with the base class name

```
#include <iostream>
class Base
{
public:
  void fun()
  {
cout<<"void"<<endl;
  }
  Int fun(int a, int b)
  {
cout<<"base"<<endl;
  }
};
class Derived: public Base
{
public:
  void fun (char c)
{
```

```
cout<<"Derived Class";
  }
};
Int main()
{
 Derived d;
d.fun('a');
d.Base::fun(2);
return(0);
}
```

6.6 Function overriding

- If derived class defines a function with same name as defined in the base class then it is known as function overriding.
- It is one of the best example of runtime polymorphism in object oriented programming
- It gives new implementation of base class method into derived class
- The number of arguments, type of arguments and the sequence of arguments should be same in base and derived classI,e the signature of both classes should be same
- the function in base class is called the overridden function and function in derived class is called as overriding function.
- In static binding, the pointer of type base class will call function inside the base class inspite it is made pointing to derived class
- The solution to this is to add virtual keyword in base class function to get run time polymorphism so that the call to the derived class can be made using the base class pointer

```
#include <iostream>
class Base
 {
```

```
public:
   void disp()
{
cout<<"Base Class";
   }
};
class Derived: public Base
{
public:
   void disp()
 {
cout<<"Derived Class";
   }
};
int main()
{
Derived obj
  Base *b = &Derived();
Obj->disp();
   return 0;
}
```

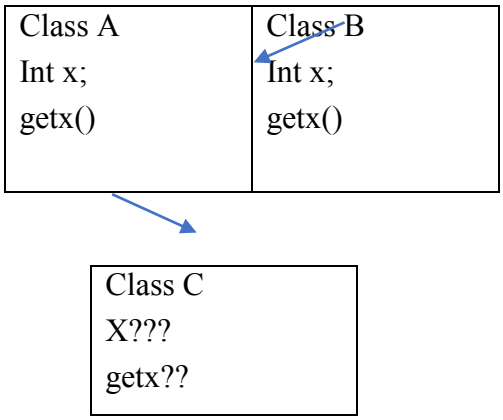
6.7 Ambiguities in multiple inheritance and multipath inheritance and solution

- When a derived class has more than one parents, an ambiguity arises in multiple inheritance and in multipath inheritance.
- The condition for ambiguity is that more than one parents class defines methods with the same name, while a class derived from both base classes has no function with this name.
- The problem is how do derived class access the correct base class function.
- The name of the function alone is insufficient, since
- the compiler can’t figure out which of the two functions is meant

Eg.

Lets say a class C inherits from both class A and class B

Class A and class B, both define a data member named int x and a data function named getx(). Ambiguity is which copy of function or data member will class C will receive?



Solution for the ambiguities:

1) Scope resolution operator

2) Virtual base class

1) Scope resolution operator

The problem of ambiguities are resolved using the scope-resolution operator to specify the class in which the function lies. Therefore

object.A::show(); // it refers to show() in class A

object.B::show(); //refers to show() in class B

2) Virtual base class

A virtual function in C++ is a member function in the base class that is defined again in derived class using the keyword virtual

The reason behind the virtual keyword is that it will make the compiler perform dynamic linkage or late binding which involves binding of data during runtime.

A single pointer to the base class is created that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, it

always executes the base class function. To resolve this issue a 'virtual' keyword is preceded at the normal declaration of the function.

When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rule for virtual function

Virtual functions must be members of some class.

It cannot be static members.

They are accessed through object pointers.

It can be a friend of another class.

A virtual function must be defined in the base class, even though it is not used.

The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.

No virtual constructor, but virtual destructor do exist

```
class base
{
public:
virtual void show()
{
        cout<<"base func"<<endl;
}
};
class derived : public base
{
public :
void show()
{
        cout<<"derived func"<<endl;
}
```

```
};
void main()
{
clrscr();
     base *b;
     derived d;
      b=&d;
b->show();
getch();
}
Output
derived func
```

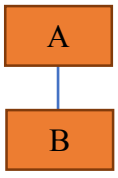
6.8 Constructor and inheritance

The inheritance of constructor depends upon the type of inheritance implemented.

6.8.1 Single inheritance

Base class constructors are called first and the derived class constructors are called next in single inheritance.

Constructor	Destructor
A()	B()
B()	A()




```
#include<iostream>
class base
{
public:
```

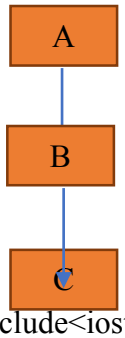
```
    base()
    {
cout<<" constructor in base class"<<endl;
    }
};
class derived:public base
{
public:
    derived()
    {
cout<<" constructor in derived class"<<endl;
    }
};
int main()
{
    derived d;
    return 0;
}
```

6.8.2 Multiple inheritance

All base class constructors are invoked first and than the derived class constructors are invoked.

Order of calling the base class constructor depends upon the type the sequence of inheritance of base class

Constructor	Destructor
A()	C()
B()	B()
C()	A()




```
#include<iostream>
class base
{
public:
   base()
   {
cout<<"constructor in base class"<<endl;
   }
};
class base1
{
public:
   base1()
   {
cout<<"constructor in base class 1"<<endl;
   }
};
class derived:public base, public base1
{
public:
   derived()
   {
cout<<”constructor in derived class"<<endl;
   }
```

```
};
int main()
{
   derived d;
   return 0;
}
```

6.8.3 Parameterized constructor

Base class parameters for parameterized constructors are supposed to be explicitly specified while calling the parameterised constructor of derived class

Base class constructor will be invoked first with the values of parameter after that the derived class constructor will be get executed with the parameter value.

```
#include<iostream.h>
#include<conio.h>
class Base
{
   int x;
   public:
   Base(int i)
   {
      x = i;
      cout<< "Base Parameterized Constructor\n";
   }
};
class Derived : public Base
{
   int y;
   public:
   Derived(int j):Base(j)
   {
      y = j;
      cout<< "Derived Parameterized Constructor\n";
```

```
    }
};
int main()
{
    Derived d(10) ;
getch();
    return 0;
}
```

6.9 Let us Sum Up

- It defines the hierarchical relationship between classes and supports the concepts of division of code
- A derived class is derived from a base class with different access control or modifiers.
- These modifiers are as follow : public inheritance, protected inheritance or private inheritance.
- Method Hiding takes place when the derived class has function with same name as that of the function in base class, then the derived class function will hide all base class function with same name even if the signatures are different
- The inheritance of constructor depends upon the type of inheritance implemented.

6.10 List of Reference

Object Oriented Programming in C++,4th Edition,RobertLafore,SAMSTechmediahttps://www.javatpoint.com/

https://www.geeksforgeeks.org/

6.11 Bibliography

The Complete Reference C, 4th EditionHerbertSehlidt,TataMcgraw Hill

The C++ Programming Language, 4th Edition,BjarneStroustrup,AddisonWesly

6.12 Unit End Exercise

- Explain the concept of multiple inheritance with an example
- Explain the solutions for ambiguities in multiple inheritance and multipath inheritance and solution
- Write a program to demonstrate hybrid inheritance

Write a short note on function overriding

❖❖❖

UNIT 4

7

INHERITANCE AND POLYMORPHISM

Unit Structure

7.0 Objective

The objective of the chapter is as follow

- To get familiar with concept of Polymorphism
- To understand the difference between static and dynamic binding
- To understand the concept of pure virtual function, virtual destructor and abstract class

7.1 Introduction

Polymorphism allows to have multiple form. This characteristic of Object Oriented Programming is explained in detail along with its types.

7.2 Polymorphism

The word polymorphism means to have multiple forms. Polymorphism occurs in classes which are related to each other by inheritance

Two types of polymorphism occur in C++:

7.2.1 Compile time Polymorphism or Static binding.

During compile time, the compiler decides which function to address if there are one or more functions with same name. Depending upon the number of arguments present in the function, the compiler decided which function to call. Function overloading implements compile time polymorphism. Compile time polymorphism is fast in terms of execution time. It is also known as early binding.

Example :

```
# include<iostream.h>
#  include<conio.h>
int area(int s);
int area(int l,int b);
float area(float r);
void main()
{
clrscr();
int a=10;
int b=20;
float c=1.2;
cout<<"area of a square "<<area(a);
cout<<"\narea of a rect "<<area(a,b);
cout<<"\n area of a circle "<<area(c);
getch();
}
int area(int s)
```

```
{
return(s*s);
}
int area(int l, int b)
{
return(l*b);
}
float area(float r)
{
return(3.14*r*r);
}
```

```
area of a square 100
area of a rect 200
 area of a circle 4.5216
```

7.2.2 Runtime Polymorphism or dynamic binding

During run time polymorphism the decision about which function to be called is taken during the runtime of the program. Function overriding implements runtime polymorphism where in the base class and the child class consist of functions with same name. Compiler decides at runtime whether the functions in the base class is supposed to be called or the function with same name in child class is supposed to be called. Run time polymorphism is slower in execution. It is also known as late binding.

```
#include <iostream.h>
#include <conio.h>
class base
{
public:
virtual void show()
{
        cout<<"base func"<<endl;
```

```
}
};
class derived : public base
{
public :
void show()
{
        cout<<"derived func"<<endl;
}
        void show1()
{
        cout<<"show1 method implemented"<<endl;
}
};
void main()
{
clrscr();
      base *b;
      derived d;
      b=&d;
      b->show();
      b->show1();
d.show();
     d.show1();
getch();
}
```

7.3 Virtual Function

Virtual functions appearto be calling a function of one class but in realityit is calling a function of another class.

When base and derived classes consist of functions with same name, the function in the base class is declared as virtual using the keyword virtual followed by its declaration.

Concept behind the virtual function involves polymorphism and late binding.

The need to make the pointer of the base class to point and refer to the rest other derived classes got resolved using virtual function.

Base class pointer pointing to derived class address used to get ignored by the compiler and instead of displaying the content inside the derived class, the content inside the base class function used to get executed every time. This was resolved using the virtual keyword in the based class.

Example:

```
base *b;
b=&d;
b->display;  // this display will refer the function of base class due to early binding.
```

Program :

```
# include <iostream>
using namespace std;
class Base
{
    public:
        void display( ) {cout<< "\n Display base ";}
        virtual void show( ) {cout; << "\n show base";}
};
class derived : public Base
{
    public;
```

```
        void display( ) {cout<< "\n Display derived";}
        void show( ) {cout<< "\ show derived";}
};
int main( )
{
    Base B;
    Derived D;
    Base *bptr;
    bptr = &D;
    bptr->display( );          // calls Base version
    bptr->show( );             // calls Derived version
    return 0;
}
```

Rule for virtual functions

- Virtual functions should be the member of some class and must be defined under base class
- Virtual function cannot be static and can be accessed using object pointer
- Virtual function can be made friend function of another class
- No virtual constructor only virtual destructor

7.4 Pure Virtual Function

Pure Virtual Function don't have any function definition and therefore also known as 'do nothing' class.

The word pure is not a keyword.To convert a virtual class into pure virtual, '=0' is supposed to be added in the virtual class declaration

Syntax :virtual void functionname()=0

The pure virtual function don't have any definition inside the base class that is why equated with zero, but it also means that the function is supposed to defined in all its derived classes.

```
class base
{
	public:
	virtual void show()
	{
		cout<<"base func"<<endl;
	}
	virtual void show1()=0;   //pure virtual
};
class derived : public base
{
	public :
	void show()
	{
		cout<<"derived func"<<endl;
	}
	void show1()
	{
		cout<<"show1 method implemented"<<endl;
	}
};
void main()
{
clrscr();
     base *b;
     derived d;
     b=&d;
     b->show();
     b->show1();
d.show();
    d.show1();
getch();
}
```

7.5 Virtual Destructor

Using the pointer to the base class, the derived class objects can be deleted using the virtual destructor

Base class pointer can store address of base class object as well as derived class object

When base class pointer points to the base class object both constructor and destructors are called.

But when base class pointer points to derived class object then constructors of both base as well as derived classes are called but destructor of only derived class is called

Complier works based on the type of pointer

This problem is resolved using the virtual destructor

```
#include <iostream.h>
#include <conio.h>
class base
{
public:
base()
{
        cout<<"base class constructor"<<endl;
}
~base()
{
        cout<<"base class destructor"<<endl;
}
};
class derived : public base
{
public :
derived()
{
```

```
        cout<<"derived class constructor"<<endl;
}
~derived()
{
        cout<<"derived class destructor"<<endl;
}
};
void main()
{
clrscr();
        base *b = new derived();
        derived *d = new derived();
      delete b;
      delete d;
getch();
}
```

7.6 Abstract Class

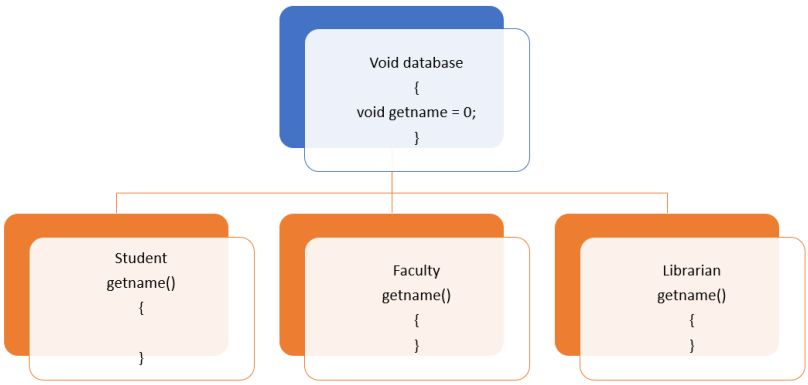
A class that contains pure virtual function is called as Abstract Class

No object of the abstract class can be created but pointer of the abstract class can be created

The usage of abstract class is to provide base class to other derived classes. All common codes of derived classes are written inside the abstract class.

All pure virtual functions inside the abstract classes are supposed to be implemented by all its derived classes or else they will become Abstract too

Eg: Base class database has a pure virtual function name getname(), this function will be compulsorily get its definition in each derived classes as student, faculty and librarian need to specify their name.




```
#include <iostream.h>
#include <conio.h>
class shape
{
        public:
        virtual void area()=0;
};
class rect : public shape
{
        int a,b;
        public:
        void area()
        {
        cout<<"enter 2 no";
        cin>>a>>b;
        cout<<a*b<<endl;
        }
};
class square : public shape
{
```

```
        int a;
        public:
        void area()
        {
        cout<<"enter 1 no";
        cin>>a;
        cout<<a*a<<endl;
        }
};
void main()
{
clrscr();
    // shape o; // error as shape is an abstract class
rect r;
     square s;
r.area();
s.area();
getch();
}
```

7.7 Interface

Behaviour or capabilities of a C++ class is described using the interface

Interface has no commitment for implementation of the described classes.

In C++ interfaces are implemented using the abstract classes.

An abstract class in created by placing atleast one pure virtual function inside the class.

7.8 Let us Sum Up

- Polymorphism is of two types 1) Compile time Polymorphism or **Static binding**and Run time Polymorphism or **Dynamic binding**.
- Virtual functions appearto be calling a function of one class but in realityit is calling a function of another class
- Using the pointer to the base class, the derived class objects can be deleted using the virtual destructor
- A class that contains pure virtual function is called as Abstract Class

7.10 List of Reference

Object Oriented Programming in C++,4th Edition,RobertLafore,SAMSTechmediahttps://www.javatpoint.com/

https://www.geeksforgeeks.org/

7.11 Bibliography

The Complete Reference C, 4th EditionHerbertSehlidt,TataMcgraw Hill

The C++ Programming Language, 4th Edition,BjarneStroustrup,AddisonWesly

7.11 Unit End Exercise

- Explain the concept of polymorphism
- Write a program to demonstrate the use of virtual function
- What are abstract classes?

Write short note on Virtual Destructor

❖❖❖

UNIT 5

8

STREAMS

Unit Structure

8.1 Objectives

This chapter would make you understand the following concepts:

- To understand file and its types.
- To learn different C++ stream classes
- To understand of hierarchy of stream classes
- To learn how to perform file-related activities using C++

- To Understand other features of C++ that are related to files, including in-memory text formatting, command-line arguments, overloading the insertion and extraction operators, and sending data to the printer.

8.2 Files

A file constitutes a sequence of bytes on the disk where a group of related or similar data is stored. File is created for permanent storage of data. A file is generally used as real-life applications that contain a large amount of data.

There are two kinds of files in a system.

1. Text files (ASCII)
2. Binary files
    - Text files contain digits, alphabetic and symbols.
    - Binary file contains collection of bytes (0's and 1's). Binary files are compiled versions of text files

8.3 Text and Binary Files

- **Binary files** contain a sequence of bytes, or ordered groupings of eight bits. A developer arranges these bytes into a format that stores the necessary information for the custom application. Binary file formats may include different types of data in the same file, for example image, video, and audio data.
- Text files contain only textual data. However, unlike binary files, they have fewer chances to become corrupted. While in a binary file may make it unreadable because of a small error.
- The basic difference between text files and binary files is that in text files various character translations are performed such as "\r+\f" is converted into "\n", whereas in binary files no such translations are performed.
- By default, C++ opens the files in text mode.
- Basic files operation in:
    1. Opening/Creating a file
    2. Closing a file
    3. Reading a file
    4. Writing in a file

We first learn the function of different file-system in c++ before proceeding for example

Name	Function
fopen( )	Open a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same asputc().
getc( )	Reads a character from a file.
fgetc( )	Same asgetc().
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file whatprintf()is to the console.
fscanf( )	Is to a file whatscanf()is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

List.1 Commonly Used File-System Functions

In the tables below we will see the various steps and operations that can (or must) be performed to use files in C++:

1) Creating or opening a file

• For writing data	
Text Files	Binary Files
ofstream out (“myfile.txt”);	ofstream out (“myfile.txt”,ios::binary);

or ofstream out; out.open("myfile.txt");	or ofstream out; out.open("myfile.txt", ios::binary);
• For Appending (adding text at the end of the existing file)	
Text Files	Binary Files

ofstream
out("myfile.txt",ios::app);
or
ofstream out;
out.open("myfile.txt", ios::app);
	ofstream out
	("myfile.txt",ios::app|ios::binary);
	or
	ofstream out;
	out.open("myfile.txt", ios::app |
	ios::binary);

• For reading data	
Text Files	Binary Files
ifstream in ("myfile.txt"); or ifstream in ; in.open("myfile.txt");	ifstream in ("myfile.txt", ios::binary); or ifstream in ; in.open("myfile.txt", ios::binary);

2) Closing Files (after reading or writing)

ofstream object "out"	**Ifstream object "in"**
out.close();	in.close();

3) Reading / Writing Data to and from files

Data	**Functions for reading file**	**Function for writing into the file**
char	get();	put();
1 word	>> (extraction operator)	<< (insertion operator)
>=1 word	getline();	<< (insertion operator)
Objects	read()	write()
Binary data	Same as above	Same as above

4) Functions that can be used to perform special tasks

Operation	function	Description
Checking the end of the file.	EOF()	Used to check eof during the reading of the file
Check if an operation fails.	bad()	Returns true if a reading or writing operation fails.
Check if an operation fails.	Fail()	Returns true in the same cases as bad (), but also in the case hat a format error happens.
Checking for the opened file.	is_open( );	Checks if the file is opened or not, returns true if the file is opened else false
Several bytes already read.	count()	Returns count of the bytes read from the file
Ignoring characters during file read.	ignore()	Ignores n bytes from the file. (get pointer is positioned after n character)
Checking the next character.	peek()	Checks the next available character, will not increase the get pointer to the next character.
Random access (only for binary files).	seekg() seekp() tellg() tellp()	In the case of binary files, random access is performed using these functions. They either give or set the position of getting and put pointers on the particular location

Example: Creating/Opening a File

```
#include<iostream>
#include<conio>
#include <fstream>
using namespace std;
int main()
{
   fstream st;                                  // Creating object of fstream class
   st.open("E:\samplefile.txt",ios::out);   // Creating new file
   if(!st)                                      // Checking whether file exist
   {
```

```
    cout<<"File creation failed";
  }
  else
  {
    cout<<"New file created";
    st.close();                    //Closing file
  }
  getch();
  return 0;
}
```

Example:Writing to a File

```
#include <iostream>
#include<conio>
#include <fstream>
using namespace std;
int main()
{
  fstream st;                          //Creating object of fstream class
  st.open("E:\samplefile.txt",ios::out);    // Creating new file
  if(!st)                              //Checking whether file exist
  {
    cout<<"File creation failed";
  }
  else
  {
    cout<<"New file created";
    st<<"Hello";                       //Writing to file
    st.close();                        //Closing file
  }
  getch();
  return 0;
}
```

Example: Reading from a File

```
#include <iostream>
#include<conio>
#include <fstream>
using namespace std;
int main()
{
   fstream st;                                  //Creating object of fstream class
   st.open("E:\samplefile.txt",ios::in);     // Creating new file
   if(!st)                                      //Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
      while (!st.eof())
      {
         st >>ch;                               //Reading from file
         cout << ch;                            //Message Read from file
      }
      st.close();                               //Closing file
   }
   getch();
   return 0;
}
```

Example: Close a File

```
#include <iostream>
#include<conio>
#include <fstream>
using namespace std;
int main()
{
   fstream st;                                      //Creating object of fstream class
   st.open("E:\samplefle.txt",ios::out);      // Step 2: Creating new file
```

```
    st.close();                                    //Closing file
    getch();
    return 0;
}
```

8.4 Stream Classes

- A stream is a flow of data. In C++ a stream is represented by an object of a special class.
- In stream class we have used the cin and cout stream objects. Different streams are used to represent different varieties of data flow. e.g. ifstream class constitutes data flow from input disk files.
- The I/O system controls file operations which are very much similar to the console input and output operations in C++ streams.
- File streams is an interface between the programs and files.
- The stream that provides data to the program is called the input stream and the one that receives data from the program is called output stream.
- The input operation is responsible for the creation of an input stream and linking it with the program and input file. Similarly, the output operation is responsible for establishing an output stream with the necessary links with the program and output file.
- Input Stream reads the data from disk files and supplies to the program using data input and receiving data from the output stream shown in the diagram.

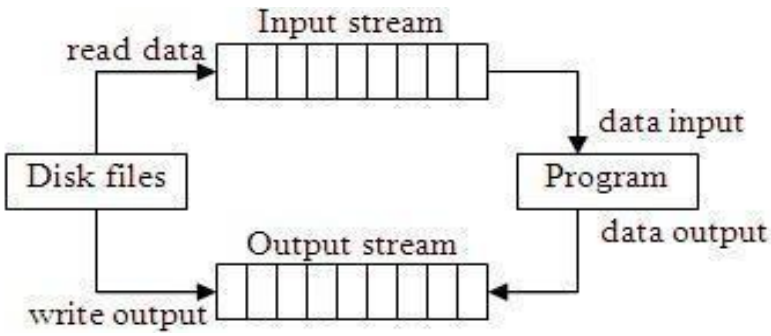


Fig.1 Input output stream

- The I/O system contains a set of classes that defines the file handling methods. These include ifstream, ofstream, and fstream.
- These classes are derived from fstreambase and form the equivalent iostream class.
- These classes, designed to manage the disk files are declared in fstream class and therefore this file is included in any program that uses these files.

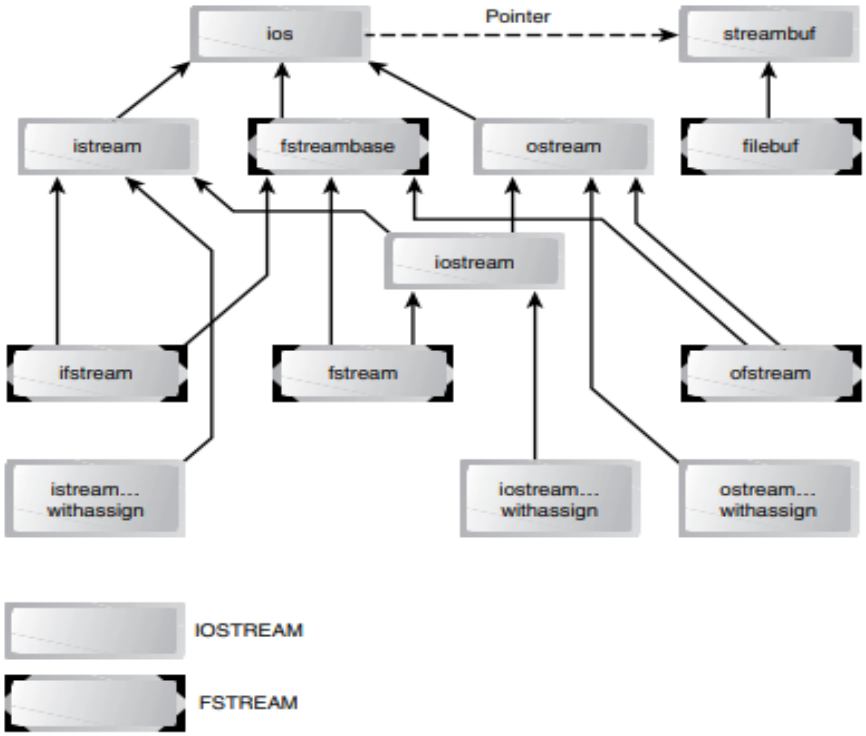


Fig.2 Stream class hierarchy.

1. **The ios class:** This class is responsible for providing all input and output facilities to all other stream classes.
2. **The istream class:** This class is responsible for handling the input stream. It provides several functions for handling chars, strings, and objects such as **to get, getline, read, ignore,**
3. **putback**, etc.
4.

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

```
int main()
{
        char x;
        // used to scan a single char
        cin.get(x);
        cout << x;
}
```

5. **The ostream class:** This class is responsible for handling output streams. It provides several functions for handling chars, strings, and objects such as **write, put**, etc.

```
#include <iostream>
using namespace std;
int main()
{
   char x;
   // used to scan a single char
   cin.get(x);
     // used to put a single char onto the screen.
   cout.put(x);
}
```

6. **The iostream:** This class is responsible for handling both input and output stream as both the istream class and istream class is inherited into it. It provides the function of both istream class and istream class for handling chars, strings, and objects such as to get, getline, read, ignore, putback, put, write, etc.

```
#include <iostream>
using namespace std;
int main()
{
   // this function display
   // ncount character from array
   cout.write(“iostreamclasses", 5); }
```

8.5 File IO using Stream classes

Detail of file stream classes

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()**and **open()**as members.
fstreambase	Provides operations common to file streams. Serves as a base for **fstream, ifstream and ofstream class**. Contains **open()** and **close()** functions.
ifstream	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg(), tellg()** functions from istream.
ofstream	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekp(), tellp()** and **write()** functions from **ostream.**
fstream	Provides support for simultaneous input and output operations. Contains open with default input mode. Inherits all the functions from **istream** and ostream classes through **iostream.**

These classes are derived directly/indirectly from the classes istream and ostream. In the stream of c++, we are using these classes: cin is an object of class istream and cout is an object of class ostream. The only difference is that we have to associate these streams with physical files. Let's see an example:

Example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Open a file

- The first operation of stream/file is opening a file for operation, generally performed on an object of one of these classes is to associate it to a real file.
- This method is known as opening a file.
- The syntax for opening a file :
  open (filename, mode);
- filename is a string : the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

os:: in	Open for input operations.
ios:: out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios:: trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

- All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

- Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios:: out
ifstream	ios::in
fstream	ios::in | ios::out

- In ifstream and ofstream classes, ios:: in and ios::out mode parameters are automatically and respectively accepted, even if a mode that does not include them is passed as a second argument to the open member function.
- For the fstream class, the default value is applied if the function is called without specifying any value for the mode parameter. If this function is called with any value in that parameter the default mode is overridden by this class, not combined in-stream class.
- In file streams, input and output operations perform independently in binary mode. Non-binary files are known as text files.
- We need to check if a file stream was successful in opening a file or not, you can do it by calling a member is_open. This open() function returns a Boolean value of true in the case that to be sure the stream object is associated with an open file or false otherwise:

```
if (myfile.is_open())
{ /* ok, proceed with output */ }
```

Closing a file

When we are finished with input and output operations on a file we need to close it so that the operating system is notified and its resources have become free. For that, we call the close () function.

```
myfile.close();
```

8.6 File pointers

A file pointer is a pointer to a structure of type FILE, the **FILE pointer** allows us to read the content of a file when we open the file in read-only mode. It automatically points at the beginning of the file, allowing us to read the file from the beginning. This pointer defines various things about the file, including its name, status, and the current position of the file.

FILE *fp;

- **Opening a File**

The fopen() function opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file. Most often, the file is a disk file. The fopen() function has this prototype:

FILE *fopen(const char *filename, const char *mode);

Where filename is a pointer to a string of characters that make up a valid filename and include a path specification. The string pointed to by mode determines how the file will be opened.

- **Closing a File**

The fclose() function closes a stream that was opened by a call to fopen() function. This function writes any data remaining in the disk buffer to the file and does a formal operating-system-level close on the file. Failure to close a stream invites all kinds of trouble, including lost data, destroyed files, and possible intermittent errors in your program. closing of a file also frees the file control block associated with the stream, making it available for reuse. There is an operating-system limit to the number of open files you may have at any one time, so you may have to close one file before opening another.

The fclose() function has this prototype:

int fclose(FILE *fp);

8.7 Error Streams

A. Un-buffered standard error stream (cerr): In C++ cerr is the standard error stream that is used to output or display the errors. This is an instance of the ostream class. As cerr in C++ is unbuffered so it is used when one needs to display the error message immediately. It does not have a buffer to store the error message and display later.

Example:

```
#include <iostream>

using namespace std;

int main()
{
   cerr << "An error occured";
   return 0;
}
```

B. buffered standard error stream (clog): This is an instance of iostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not filled and error message will be displayed on the screen.

Example:

```
#include <iostream>
using namespace std;
  int main()
{
   clog << "An error occured";

   return 0;
}
```

8.8 Random File Access

- Random file access enables us to read & write any data in our disk file, in random access we can quickly search for data, modify data & delete data. data fseek( ) and **Random-Access I/O**
- Random-access read and write operations using an I/O system with the help of fseek(), which sets the file position indicator. Its prototype is shown here:

int fseek(FILE *fp, longnumbytes, intorigin);

- Here, fp is a file pointer returned by a call to fopen().numbytes is the number of bytes from the origin that will become the new current position, and origin is one of the following macros:

Origin	**Macro Name**
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

8.9 Manipulators

- Manipulators are formatting instructions inserted directly into a stream.
- Manipulators are helping functions specifically designed to be used in conjunction with the input output stream.
- It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.
- For example, if we want to print the hexadecimal value of 100 then we can print it as:

cout << setbase(16) << 100

Types of Manipulators

1. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.

endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes the output stream.

ws: It is defined in istream and is used to ignore the whitespaces in the string sequence.

ends: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

flush: It is also defined in ostream and it flushes the output stream i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same but may not appear in real-time.

Example:

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;
```

```
int main()
{
   istringstream str("          Programmer");
   string line;
   // Ignore all the whitespace in string
   // str before the first word.
   getline(str >> std::ws, line);

   // you can also write str>>ws
   // After printing the output it will automatically
   // write a new line in the output stream.
   cout << line << endl;

   // without flush, the output will be the same.
   cout << "only a test" << flush;

   // Use of ends Manipulator
   cout << "\na";

   // NULL character will be added in the Output
   cout << "b" << ends;
   cout << "c" << endl;

   return 0;
}
```

Output:

```
Programmer
only a test
abc
```

2. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill ('*'), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.

For Example, you can use following manipulators to set minimum width and fill the space with any character you want:

std::cout << std::setw (6) << std::setfill ('*');

- **Some important manipulators in <iomanip> are:**

setw (val): It is used to sets the field width in output operations.

setfill (c): It is used to fill the character 'c' on output stream.

setprecision (val): It sets val as the new value for the precision of floating-point values.

setbase(val): It is used to set the numeric base value for numeric values.

setiosflags(flag): It is used to sets the format flags specified by parameter mask.

resetiosflags(m): It is used to resets the format flags specified by parameter mask.

- **Some important manipulators in <ios> are:**

showpos: It forces you to show a positive sign on positive numbers.

noshowpos: It forces not to write a positive sign on positive numbers.

showbase: It indicates the numeric base of numeric values.

uppercase: It forces uppercase letters for numeric values.

nouppercase: It forces lowercase letters for numeric values.

fixed: It uses decimal notation for floating-point values.

scientific: It uses scientific floating-point notation.

hex: Read and write hexadecimal values for integers and it works the same as the setbase(16).

dec: Read and write decimal values for integers i.e. setbase(10).

oct: Read and write octal values for integers i.e. setbase(10).

left: It adjusts output to the left.

right: It adjusts output to the right.

Example:

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

int main()
{
```

```
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;

    // We can use setbase(16) here instead of hex

    // formatting
    cout << hex << left << showbase << nouppercase;

    // actual printed part
    cout << (long long)A << endl;

    // We can use dec here instead of setbase(10)

    // formatting
    cout << setbase(10) << right << setw(15)
         << setfill('_') << showpos
         << fixed << setprecision(2);

    // actual printed part
    cout << B << endl;

    // formatting
    cout << scientific << uppercase
         << noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

Output:

```
0x64
_______+2001.53
2.014552646E+05
```

8.10 Overloading Insertion and extraction operators

- This is a powerful feature of C++. It lets you treat I/O for user-defined data types in the same way as basic types like int and double.

- For example, if you have an object of class crawdad called cd1, you can display it with the statement

cout << "\ncd1=" << cd1;

- C++ can input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.
- The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.
- Here, it is important to make the operator overloading function a friend of the class because it would be called without creating an object.
- The following example explains how the extraction operator >> and insertion operator <<.

Example:

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
Distance(int f, int i){
      feet = f;
      inches = i;
    }
    friend ostream &operator<<( ostream &output,
      const Distance &D ) {
      output << "F : " << D.feet << " I : " << D.inches;
      return output;
    }

    friend istream &operator>>( istream  &input, Distance &D ) {
```

```
        input >> D.feet >> D.inches;
        return input;
    }
};

int main()
{
  Distance D1(11, 10), D2(5, 11), D3;

  cout << "Enter the value of object : " << endl;
  cin >> D3;
  cout << "First Distance : " << D1 << endl;
  cout << "Second Distance :" << D2 << endl;
  cout << "Third Distance :" << D3 << endl;


  return 0;
}
```

Output:

- Enter the value of object :
- 70
- 10
- First Distance:F : 11 I : 10
- Second Distance:F : 5 I : 11
- Third Distance:F : 70 I : 10

8.11 Summary

- In this chapter we briefly examined the hierarchy of stream classes and showed how to handle various kinds of I/O errors. Then we saw how to perform file I/O in a variety of ways.
- Files in C++ are associated with objects of various classes, typically ofstream for output, ifstream for input, and fstream for both input and output.
- Member functions of these or base classes are used to perform I/O operations. Such operators and functions as <<, put(), and write() are used for output, while >>, get(), and read() are used for input.

- The read() and write() functions work in binary mode, so entire objects can be saved to disk no matter what sort of data they contain.
- A check for error conditions should be made after each file operation. The file object itself takes on a value of 0 if an error occurred. Also, several member functions can be used to determine specific kinds of errors.
- The extraction operator >> and the insertion operator << can be overloaded so that they work with programmer-defined data types. Memory can be considered a stream, and data sent to it as if it were a file.

8.12 Reference for further reading

Reference Books:

1. Object Oriented Programming in C++,4th Edition,Robert Lafore,SAMS Techmedia
2. The C++ Programming Language, 4th Edition,Bjarne Stroustrup, Addison Wesley

Web References:

1. www.geeksforgeeks.org
2. www.javatpoint.com

8.13 Unit End Exercises

1. Explain the difference betweenText and Binary Files
2. Name three stream classes commonly used for disk I/O.
3. Write a statement that writes a single character to an object called fileOut, which is of class ofstream.
4. Define what current position means when applied to files.
5. True or false: A file pointer always contains the address of the file.
6. Write the declarator for the overloaded >> operator that takes output from an object of class istream and displays it as the contents of an object of class Sample.

❖❖❖

UNIT 5

9

EXCEPTIONS

Unit Structure

9.1 Objectives

This chapter would make you understand the following concepts:

- Exceptions provide a convenient, uniform way to handle errors that occur within classes.
- To learn different kinds of exceptions.
- To learn how to prevent Exceptions provide a convenient, uniform way to handle errors that occur within classes.

9.2 Error handling

Logical errors and syntactic errors are two most common types of error in a C++ programming language. The logic errors occur due to a poor understanding of the problems in a particular area of subject. the syntax error arises due to poor understanding of the programming language.

Errors can be categorized into two types.

1. Compile Time: When you start compiling a program through compiler Exception is caught.
2. Run Time: After compilation when the program starts running from memory the run time error is thrown.

9.3 Exceptions

In C++ Exceptions class identified an error while problems arise during compiling and running a program. Exceptions are errors that occur at runtime. The errors are caused by a wide variety of exceptional conditions, for example running out of memory, unable to open a file, trying to initialize an object which is created by a class, or using an out-of-bounds index to a vector.

9.3.1 Need of exceptions:

An exception is needed for successfully running a program or getting and desiring output, during runtime programs often throw an error by returning a particular value from the function. For example, disk-file functions often return NULL or 0 to show an error

9.3.2 Exception mechanism:

In C++ Programming, objects are created using class, it means that they interact with each other, during this interaction some problem occurs at this time program detect an error in a try block and inform the exception handler to catch this exception and display. try block throw and exception. Problem detected in the try block will be caught in the catch block.

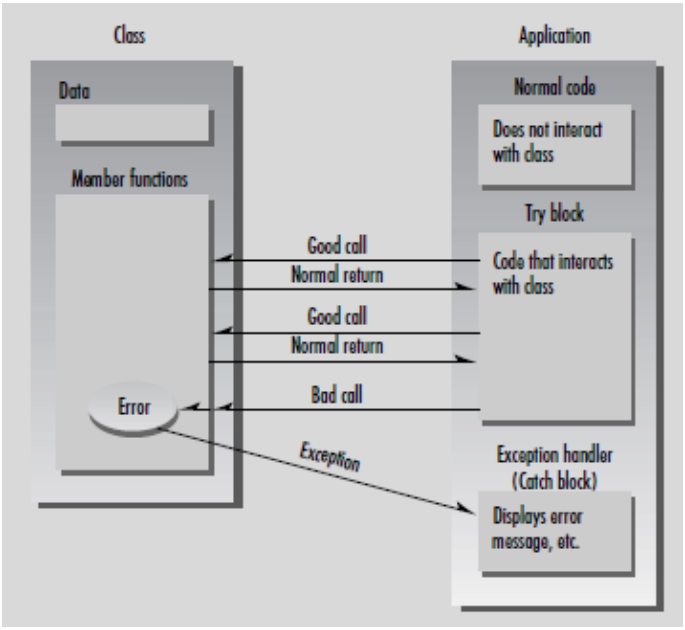


Fig.1 The exception mechanism

Syntax:

```
try
 {
     //code1
 } catch( ExceptionName e1one )
{
   // catch1
} catch( ExceptionName e2two )
{
   // catch2
} catch( ExceptionName enth )
 {
   // catch3
}
```

Example:

```
#include <iostream>
using namespace std;
const int MAX = 3;                              //stack holds 3 integers
```

```
class Stack
{
    private:
    int st[MAX]; //array of integers
    int top; //index of top of stack
    public:
    class Excp                    //exception class for Stack
    {                              //note: empty class body
     };
    Stack()
    {
        top = -1;
    }
    void push(int var)
    {
        if(top >= MAX-1)          //if stack full,
        throw Excp();             //throw exception
        st[++top] = var;          //put number on stack
    }
   int pop()
    {
        if(top < 0)               //if stack empty,
        throw Excp();             //throw exception
        return st[top--];         //take number off stack
    }
};
```

9.3.3 Multiple catch Statements

In C++ we can use more than one catch block with a try block. each catch must catch a different type of exception.

Example, this program catches both integers and strings types in c++.

```
#include <iostream>
using namespace std;
void mulcatch(int testdigit)
```

```
{
try
        {
        if(test)
            throw testdigit;
        else
            throw "Value is zero";
        }
        catch(int i)
        {
             cout << "Caught Exception: " << i << '\n';
        }
        catch(const char *str)
        {
               cout << "Caught a string exception: ";
               cout << str << '\n';
        }
}
int main()
{
        cout << "Start\n";
        mulcatch(0);
        mulcatch(1);
        mulcatch(0);
        mulcatch(2);
        cout << "End of program";
        return 0;
}
Output:
Start
Caught Exception : 0
Caught Exception : 1
Caught a string exception: Value is zero
Caught Exception : 2
End of program
```

9.4 Throwing and catching Exceptions

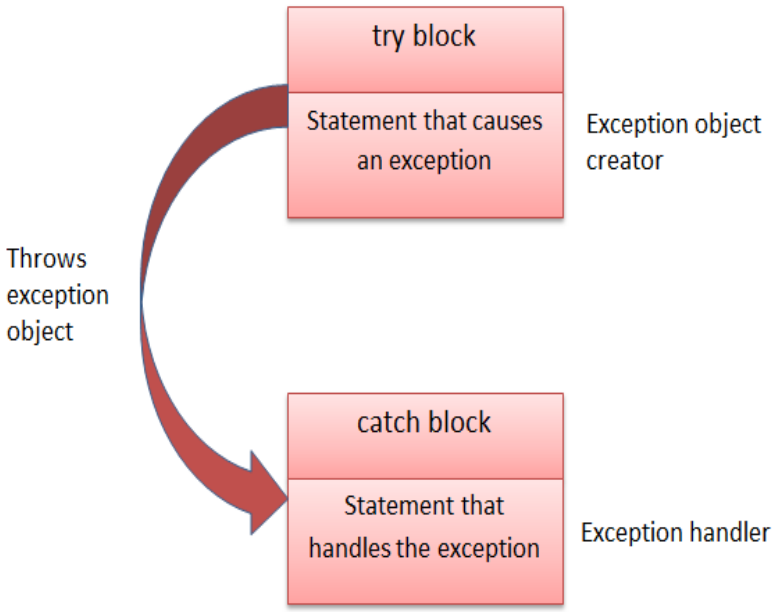


fig.3 Throwing and catching Exceptions

- throws − When an exception or error is found in a program during compilation or running, it's a throw using the throw statement in the try block of a program.
- catch block− after throwing an exception in a program catches an exception or error with an exception handler class at the place in a program where you want to handle and operate the problem. The catch keyword indicates the catching of an exception or error in a C++ program.
- try block −. in this block logical code is kept for identifying a problem. The keyword try is used to introduce a block of the statement which may generate an exception. It's followed by one or more catch blocks.

```
// exceptions
#include<iostream>
using namespace std;
itry {
  int size = 15;
  if (size > 18) {
        cout << "Access granted - you are fit.";
  } else {
```

```
        throw (size);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 to fit.\n";
  cout << "Size is: " << myNum;
}
```

- Exceptions is throw back to main method, which has a try..catch block
- Different catch blocks for different exceptions.
- throwing an exception will end the method, it will affect the flow of code.
- If an exception occurred in the middle of the method, and the code below it can't run.
- If that exception happened, then you would need to either enclose the whole section in a try, catch block or throw an exception.
- In C++ exception class, catching multiple exceptions and handling them separately by supplying multiple catch blocks in a program.
- If code may throw a checked exception, you have two choices.
    1. One is to catch the exception.
    2. The other throw exception Name in the method header.
- For throwing an exception, the throw operator is used and gives it an exception object as an argument.

The general form of the throw statement:

Throw exception;

Example:

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

```
cout << "Example for Throwing of an Exception t\n";
try
{
    cout << "We are Inside a try block\n";
    throw 100;                                    // throw an error
    cout << "Not executed";
}
catch (int i)
{                                                 // catch an error
    cout << "Caught an exception: ";
    cout << i << "\n";
}
    cout << "Process End";
    return 0;
}
```

Output:

```
Example for Throwing of an Exception
We are Inside a try block
Caught an exception: 100
Process End
```

- Only two of the three statements are executed: the cout statement and the throw.
- Control passes to the catch expression, and the try block is terminated.
- In the below example, we change the type in the catch statement to double data type, the exception will not be caught and abnormal termination will occur.

Example:

```
// This example will not work.
#include <iostream>
using namespace std;
int main()
{
   cout << "Display of cout\n";
   try
```

```
    {
      cout << "we are Inside try block\n";
      throw 100;
      cout << "This not executed";
    }
    catch (double i)
    {
     cout << "Caught an exception: ";
     cout << i << "\n";
    }
cout << "End";
return 0;
}
```

Output:

Display of cout
we are Inside try block
terminate called after throwing an instance of ‘int’

9.5 Custom Exceptions

- You can make your own exception class with help of std::exception.
- Custom exception class directly inherits from std::exception class .

```
#include <iostream>
#include <exception>
struct MyExcep : public std::exception
{
        const char * what () const throw ()
  {
        return "C++ Exception class";
  }
}
int main()
{
        try
  {
        throw MyExcep();
  }
```

```
  catch (MyExcep& e)
  {
        std::cout << "MyExcep caught" << std::endl;
         std::cout << e.what() << std::endl;
  }
  catch (std::exception& e)
  {
       // Other errors if any
  }
}
```

9.6 Built-in exceptions

- C++ provides a different type of built in exceptions.
- The exception class is the base class for all exception.

Exceptions derived directly from exception class:

bad_alloc	Failure of memory allocation
bad_cast	Incorrect use of dynamic_cast
bad_exception	Thrown by unexpected handler
bad_function_call	empty function is called
bad_typeid	Thrown by typeid function
bad_weak_ptr	thrown by shared_ptr class constructor
ios_base::failure	Base class for all the stream exceptions
logic_error	Base class for some logic error exceptions
runtime_error	Base class for some runtime error exceptions

Exceptions derived indirectly from exception class through logic_error:

domain_error	Thrown when an error of function domain happens
future_error	Reports an exception that can happen in **future** objects(See more info about **future** class)
nvalid_argument	Exception is thrown when invalid argument is passed
length_error	Is thrown when incorrect length is set
out_of range error	Is thrown when out of range index is used

Exceptions derived indirectly from exception class through runtime_error:

overflow_error	Arithmetic overflow exception
range_error	Signals range error in computations
system_error	Reports an exception from OS
underflow_error	Arithmetic underflow exceptions

Example 1:

Throw a custom exception object-1

```
#include<iostream>
using namespace std;
Class Problem
{
        public:
        Problem(const char* pStr=”There’s a problem”):pMessage(pStr)
        {
        }
        const char* what() const
        {
            return pMessage;
        }
       private:
```

```
        const char* pMessage;
};
```

Example 2:

Throw a custom exception object-2

```
#include<iostream>
using namespace std;
int main()
{
    for(int i=0;i<2;i++)
    {
        try
        {
            if(i==0)
                throw Problem();
            else
                throw Problem("No one seen the problems");
        }
        catch(const Problem& t)
        {
            cout << endl << "Exception:" << t.what();
        }
    }
    return 0;
}
```

9.7 Summary

- Exceptions are a procedure for handling C++ errors in a systematic, OOP-oriented way.
- An exception is typically caused by a defective statement in a try block that operates on objects of a class.

- The class member function finds the error and throws an exception, which is caught by the program using the class, in exception-handler code following the try block.

9.8 References for further reading:

1. Object Oriented Programming in C++,4th Edition,Robert Lafore,SAMS Techmedia

2. The C++ Programming Language, 4th Edition,Bjarne Stroustrup, Addison Wesley

9.9 Unit End Exercises

1. Explain the error handling mechanism with the help of example?

2. Write a C++ program to demonstrate throwing and catching exceptions?

3. What are multiple exceptions? explain with examples.

4. To implement a program to handle Stack Full and Stack Empty Exception class Stack using its own Exception class.

❖❖❖

UNIT 6

10

CASTING, HEADER FILES & LIBRARIES & NAMESPACES

Unit Structure

10.0 Objectives

The objectives of this chapter are to:

- Explain the concepts of Casting and its types
- Explain how to create different types of libraries & header files
- Explain the concept of Namespaces and how to use it

10.1 Introduction

- This chapter deals with C++ concepts such as type conversion, header files & libraries and namespaces.
- These are considered to be some advanced concepts.
- Some of the concepts here are new to C++ and were not present in C

10.2 Casting

- The concept of casting (or Type Casting in general) is related to (data)type conversion. Type conversion is converting an expression of one type into another type. Example converting a float type expression to integer type.
- There are two types of conversions in C++:
  - Implicit Conversion
  - Explicit Conversion
- **Implicit Conversion** – This type conversion is carried out by the C++ compiler automatically.

Example:

```
short a = 10;
int b = a;
```

- Here, variable a is declared to be a short integer.
- After assigning a to b, the value of a is promoted to integer from short by the compiler automatically without us having to tell it to do so explicitly.
- **Explicit Conversion** – Not all conversions can be automatically done by the compiler. Explicit Conversions allow us to force conversions when normally it cannot be carried out by the compiler automatically (or implicitly).

- Here onwards we refer to performing Explicit Conversions as Type Casting.
- **Definition of a cast:A cast is a special operator that forces conversion of one data type into another. A cast is a unary operator and has the same precedence as any other unary operator.**
- C++ defines **five** casting operators.
  - The traditional C-style cast
  - dynamic_cast
  - const_cast,
  - reinterpret_cast, and
  - static_cast.
- The traditional C-style cast : The casting is performed by putting the desired type in parenthesis to the left of the variable we want to cast.

```
Example:
double a = 10.5;
int b;
b  = (int)  a;
```

Output : 10

10.2.1 Static Casts

- The static_cast operator is the simplest type of cast, it is a replacement for the original cast operator. It simply performs a non-polymorphic cast.
- It is a compile time cast. It performs implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions (or implicit ones).
- It can be used for any standard conversion. No run-time checks are performed.
- Its general form is **static_cast<type> (expr)**
- For Example:

```
double a = 10.5;
int b;
b  =static_cast<int>  a;
cout<< b;
```

Output : 10

10.2.2 Const Casts

- It is used to cast away the constness of variables. In other words, it is used to explicitly override const and/or volatile in a cast.
- The general form of const_cast is shown here: const_cast<type> (expr)
- The following example removes the const-ness of a pointer

void cubeval (constint *val)
{
int *a;
// cast away const-ness.
a = const_cast<int *> (val);
*a = *val * *val* *val;
// modify object through val
}
	int main()
	{
	int c = 5;
	cout<< "c before call: " << c;
	cout<<endl;
	cubeval(&c);
	cout<< "c after call: " << c;
	return 0;
	}

Output:

c before call: 5

c after call: 125

10.2.3 Dynamic Casts

- dynamic_cast may be used to cast one type of pointer into another or one type of reference into another.
- The dynamic_cast performs a run-time cast. It verifies the validity of a cast.
- If the cast is invalid at the time dynamic_cast is executed, then the cast fails, i.eIf the cast cannot be made, the cast fails and the expression evaluates to null.
- The general form of dynamic_cast is shown here:

dynamic_cast<target-type> (expr)

- For example, consider two classes B and D, with D derived from B.

- Since a base pointer can always point to a derived object, a dynamic_cast can always cast a D* pointer into a B* pointer. Consider the code below

```
Base *bp, b_ob;
Derived *dp, d_ob;
bp = &d_ob; // base pointer points to Derived object
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
if(dp) cout<< "Cast OK";
```

Output: Cast OK

- Here, the cast from the base pointer bp to the derived pointer dp works because bp is actually pointing to a Derived object. Thus, this fragment displays Cast OK.
- In the next fragment, the cast fails because bp is pointing to a Base object and it is illegal to cast a base object into a derived object.

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout<< "Cast Fails";
```

Output: Cast Fails

- Because the cast fails, this fragment displays Cast Fails.
- A dynamic_cast can cast a B* pointer into a D* pointer only if the object being pointed to actually is a D object.
- In general, dynamic_cast will succeed if the pointer (or reference) being cast is a pointer (or reference) to either an object of the target type or an object derived from the target type. Otherwise, the cast will fail.
- If the cast fails, then dynamic_cast evaluates to null if the cast involves pointers.

10.2.4 Reinterpret Casts

- **Reinterpret Cast** operator converts one type into a fundamentally different type.
- Simply put, it allows any pointer to be converted into any other pointer type. Also, it can change a pointer into an integer and vice versa.
- It does not check if the pointer type and data pointed by the pointer is the same or not.

- The general form of **reinterpret_cast** is shown here:

reinterpret_cast<type> (expr)

- Example

```
int main()
{
int* a= new int(78);
   char* b = reinterpret_cast<char*>(a);
cout<< *a <<endl;
cout<< *b <<endl;
cout<< a <<endl;
cout<<ch<<endl;
   return 0;
}
```

Output:

78

N

0x1c2ae70

N

10.3 Libraries and Header files

- Header files & Libraries provide a way to make code reusable. If we have some code that we want to share or reuse, we can create a header or library with it and link the library with any application that needs it.
- **Header Files**
  - They tell the compiler how to call some functionality, they contain the function prototypes.
  - They have the extension ".h"
  - Example: GRAPHICS.H
- **Libraries or a Library** is where the actual functionality is implemented i.e. they contain the function definition.

- Libraries are of two types :
  - Static
  - Shared or Dynamic
- Static Library
  - Also called as archive library
  - A static (archive) library generally has a ".a" extension
  - With a static library, its objects are generally contained in the object code linked with an end-users application, and then becomes part of that executable.
  - **Compile time -** These libraries are used at *compile time* meaning the library should be present in the correct location when the user wants to compile the C++ program.
- Dynamic Library
  - Also called as **Shared** library
  - A dynamic library generally has a ".so" extension
  - With a dynamic (shared) library, objects within the library are not linked into the program's executable file, but are loaded by the compiler when required for execution.
  - **Run time :**These libraries are used at *run-time*i.e, the code is compiled without using these libraries and these are linked at compile time to resolve undefined references. It is then distributed to the application so that application can load it at run time.

10.3.1 Creating Libraries

We will use the following codes to work with libraries.

// calc.cpp
#include <math.h>
#include "MYLIBARY.h"
intcalcCube(int d)
{
return (d * d * d);
}
	// main.cpp
	#include <iostream.h>
	#include "MYLIBARY.h"
	int main()
	{
	int d = 5;
	cout<<"Cube of " <<d<< " = "
	<<calcCube(d);
	return 0;
	}

// MYLIBARY.h
intcalcCube(int);	

10.3.1.1 Creating Static Library

1. Compile the individual cpp files as shown below

```
gcc -c main.cpp                 //Creates main.o
gcc -c calc.cpp                 //Creates calc.o
gcc -o out calc.omain.o         //Creates executable out
start out.exe                           //Gives output 125
```

2. Building a library using "ar"

```
arcrvlibcalc.acalc.o        //Creates a library
```

3. Create executable from the library libcalc.a

```
gcc -o final main.olibcalc.a
```

10.3.1.2 Creating Dynamic Library

1. Compile the individual cpp files with fPICas shown below

   ```
   gcc-Wall -fPIC-c main.cpp     //Creates main.o
   gcc-Wall -fPIC-c calc.cpp     //Creates calc.o
   ```

   - "-Wall" is a warning option, it enables warnings for many common errors
   - "-fPIC" option will enable *"position independent code"*generation, it is a requirement for shared libraries.

2. Building a library using "

   ```
   gcc -shared -o libcalc.so calc.o
   ```

   - "-shared" creates a shared object libcalc.so which can then be linked with other objects to form an executable.

3. Create executable from the library **libcalc.so**

   ```
   gccmain.o ./libcalc.so -lm
   ```

   - Here, the primary object **main.o**, has dependencies on **calc.o,** so the linker is invoked as shown above

10.3.2 Creating header files

- A header file usually has ".h" extension.
- A header file in C/C++ contains:
  1. Function definitions
  2. Data type definitions
  3. Macros

- A header file is always used with a preprocessor directive called **#include,** a preprocessor directive responsible for instructing the C/C++ compiler that these(header) files need to be processed before compilation.
- Creating a header file (cube.h) for calculating cube of a number:
    1. Write the code(logic) and save with ".h" extension
    2. Use the header file on top in program you want to use it
    3. Compile & Run the Program

// calcCube.cpp
#include <iostream.h>
#include "cube.h"
int main()
{
int d = 5;
cout<<"Cube of " <<d<< " = " <<calcCube(d);
return 0;
}
	// cube.h
	intcalcCube(int d)
	{
	return (d * d * d);
	}

10.4 Namespaces

- Namespace is associated with scope. Scope defines and affects the visibility of local & global variables.
- A namespace is a declaration that provides a scope to the identifiers inside it. They are used to localize the names of identifiers to avoid name collisions.
- The members of a namespace belong to the same scope and can refer to each other without special notation, whereas access from outside the namespace requires explicit notation.

10.4.1 Defining a Namespace

- A namespaces is defined using the keyword **namespace** followed by the namespace name.
- defining a namespace allows you to partition the global namespace by creating a declarative region, a namespace defines a scope
- The general form of namespace is shown by:

```
namespace name {
// declarations
}
```

- For example:

```
namespacemynewNamespace
{
int x, y;
}
```

- In this case, the variables a and b are normal variables declared within a namespace called mynewNamespace.

10.4.2 Accessing Namespace objects

- From within their namespace normally, variables can be accessed directly with their identifier
- From outside their namespace variables can be accessed using the **scope resolution operator (::)**
- **Example:**

```
using namespace std;
// Variable created inside namespace
namespace one
{
int a = 100;
}
int main()
{
   // Local variable
int a = 200;
cout<<a;         // outputs value local variable
   // These variables can be accessed from outside the
   //namespace using the scope operator ::
cout<< one::a << '\n';
    // outputs value local variable with namespace one
return 0;
}
```

Output:
200
100

10.4.3 The *using* directive

- Frequent references to members of a namespace by having to specify the scope resolution operator makes programming a tedious job.
- The using namespace directive allows us to avoid prepending of namespaces with the scope resolution operator. The **using**directive tells the compiler that the subsequent code uses names in the specified namespace.
- The using statement has these two general forms:
    - using namespace name;
    - using name::member;
- Example

```
#include <iostream>
using namespace std;
// first name space
namespace first_space {
  void func() {
cout<< "Inside first_space";
  }
}
// second name space
namespace second_space {
  void func() {
cout<< "Inside second_space";
  }
}
```

```
using namespace first_space;
int main () {
// calls function from first name space.
func();
  return 0;
}
```

Output: Inside first_space

10.4.4 Unnamed Namespaces

- It is a special type of namespace
- It allows you to create identifiers that are unique within a file.
- Unnamed namespaces are also called anonymous namespaces.

- They have this general form:

```
namespace {
// declarations
}
```

10.4.5 Discontiguous Namespaces

- A namespace can be defined in multiple parts. Hence, a namespace is made up of the sum of its separately defined parts.
- The separate parts of a namespace can be spread over multiple files.
- Example

```
#include <iostream>
using namespace std;
namespace IDOL {
  {
int a;
  }
}
namespace IDOL {
  {
int b;
  }
}
```

```
int main () {
  IDOL :: i = IDOL :: j = 5;
  // refer to IDOL specifically
cout<<IDOL: :i * IDOL:: j << "\n";
  // useIDOL namespace
 using namespaceIDOL;
cout<< i * j;
   return 0;
}
```

Output:
25
25

10.4.6 Nested Namespaces

- Namespaces allow nesting, i.e - it is possible to define one namespace inside another name space
- Example

```
#include <iostream>
using namespace std;
namespace N1 {
```

```
int main () {
  N1 :: i = 10;
  // N2 :: j = 20;
```

```
int i;
namespace N2 {
 // a nested namespace
int j;
   }
}
```

```
   //Error, N2 is not in view
   N1 :: N2 :: j = 20; // this is right
cout<< N1 :: i << " ";
cout<< N1 :: N2 :: j << "\n";
  // use N1
  using namespace N1;
  /* Now that N1 is in view, N2 can
      be used to refer to j. */
cout<< i * N2 :: j;
    return 0;
}
```

Output:

10 20

200

10.4.7 The *std* Namespace

- Standard C++ defines its entire library in its own namespace called std.
- This is the reason why most of the programs in C++ include the following statement **using namespace std;**
- This causes the std namespace to be the current namespace giving direct access to the names the functions and classes defined without having to use std:: everytime.
- Example : consider the following example using and without using std namespace

```
// using namespace std
#include <iostream>
using namespace std;
int main () {
cout<<"Happy Learning !";
   return 0;
}
```

```
// without using namespace std
#include <iostream>
using std::cout;
int main () {
cout<<"Happy Learning !";
   return 0;
}
```

10.5 Review Question

1. What is type casting? Explain the two main types of type conversion.
2. Explain the different types of casts with examples
3. Explain the difference between libraries and header files.
4. What are Libraries? What are its types?
5. Explain the process of creating static & dynamic libraries.
6. Explain with example creating and using a header file.
7. Explain the concept of a Namespace in detail

10.6 References

Books

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, AddisonWesly
3. Absolute C++,4th Edition, Walter Savitch,Pearson Education

Web References

1. https://www.geeksforgeeks.org/difference-header-file-library/
2. https://docs.oracle.com/cd/E19205-01/819-5267/bkamn/index.html
3. https://www.bogotobogo.com/cplusplus/libraries.php
4. https://data-flair.training/blogs/header-files-in-c-cpp/

❖❖❖

UNIT 6

11

GENERIC PROGRAMMING, TEMPLATES & STL

Unit Structure

11.0 Objectives

The objectives of this chapter are to:

- Explain the concepts of Generic Programming
- Explain Templates & associated concepts
- Explain the concept of STL

11.1 Introduction

- In this chapter we continuing our journey in the advanced concepts in C++
- This chapter deals with C++ concepts of Generic Programming & Templates.
- Templates are one of C++'s most sophisticated and high-powered features. Using templates we will create generic functions and classes.

11.2 Generic Programming

- Generic programming is a kind of programming during which algorithms are written in terms of the data types that will be specified later in the development and instantiated as per need for specific types and provided as parameters.
- Simply put it refers to the programming model where ***algorithms*** and ***functions*** are written in terms of 'types' so that it works on all data types and not just for one. It enables the programmer to write a general algorithm which will work with all data types.
- This approach allows us to write common functions or types that differ only within the set of types on which they operate when used, thus reducing duplication. Such software entities are referred to as *generics.*
- The method of Generic Programming is implemented to increase the efficiency of the code.
- The concept of Genericity is utilized and supported in different ways by programming languages
- Arrays and structs are often viewed as predefined generic types. Every usage of an array or struct type instantiates a new base type, or reuses a previous instantiated type. Element types such as Array and struct are parameterized types, which are used to instantiate the corresponding generic type. All this is often usually built-in in the compiler and therefore the syntax differs from other generic constructs.
- The advantages of Generic Programming are
    1. Code Reusability
    2. Avoid Function Overloading
    3. Once written it can often be used for multiple times and cases.

- Generics are often implemented in C++ using Templates. Template could be a simple and yet very powerful tool in C++. The straightforward idea is to pass data type as a parameter in order that we don't have to write the exact same or equivalent code for various data types.
- For example, a software company may have a function sort() for different data types. Instead of writing and maintaining the multiple codes, we will write one function sort() and pass data type as a parameter.

11.3 Templates

- Templates support generic programming. It allows developing reusable software components such as functions, classes, etc. supporting different data types insidethe same frame work.
- A template in C++ can be used to create a family of classes or functions, it allows the construction of a family of template functions and classes to perform the same operation on different data types.
- The templates declared for functions are called class templates. They perform appropriate operations and rely on the data type of the arguments passed to them.

11.3.1 Class Templates

- Class templates are declared to operate on different data types.
- A class template specifies how individual classes can be constructed similar to a normal class definition.
- These classes model a generic class which support similar operations for various data types.
- The general form of a using a class template is:

```
Template <class T>
class classname
{
T member1;
T member2;
…
…
```

```
public:
T functionName(T arg);
…
..
};
```

- As shown above, a class template is created like any other class, except the fact that it is preceded with the declaration ***Template <class T>*** which specifies that what follows is a class template.T is the argument for template and is a kindof placeholder for the data type used.
- To create object of a class template, we have to define the data type within a <> when creation. The general format is :

  className<dataType> classObject;

- Example:
  - className<int> classObject;
  - className<float> classObject;
  - className<string> classObject;
- Template example : Addition using Class Template

```
#include <iostream>
using namespace std;
template <class T>
class IDOL
{
private:
        T arg1, arg2;
public:
        IDOL(T n1, T n2)
        {
                arg1 = n1;
                arg2 = n2;
        }

        void displayResult()
        {
```

```
            cout << "Numbers are: " <<arg1<< " and " <<arg2<< "." << endl;
            cout << "Addition is: " << add() << endl;
            cout << "Subtraction is: " << subtract() << endl;


        }
        T add()
{
 return arg1 + arg2;
}
};
int main()
{
        IDOL<int> intAdd(5, 4);
        IDOL<float> floatAdd(9.1, 4.5);

        cout << "Int results:" << endl;
        intAdd.displayResult();

        cout << endl << "Float results:" << endl;
        floatAdd.displayResult();

        return 0;
}
```

Output

Int results: Numbers are: 5 and 4. Addition is: 9	Float results: Numbers are: 9.1 and 4.5. Addition is: 13.6

- Explanation:
    - The above program defines a class template **IDOL.** .
    - The class contains two private members of type T: arg1 & arg2, and a constructor to initalize the members.
    - It contains public member functions to calculate the addition of the numbers which returns the value of data type defined by the user & a function displayResult() to display the output to the screen.

- The main() function defines two different IDOL objects intAdd and floatAdd created for data types: int and float respectively. Their values are initialized using the constructor.
- Kindly note that we used<int> and <float> while creating the objects. These tell the compiler of the data type used for the class creation.
- This creates a class definition for both int and float, which are then used accordingly.
- Then, displayResult() of both objects is called which performs the Addition operation and displays the output.

11.3.2 Function Templates

- A function that successfully works for every C++ data type is known as a generic function.
- Arguments used as generic data types In templates and they can handle a variety of data types.
- Templates are a way to help the programmer to declare a group of functions and/ or classes. Templates when they are used with functions they are called normally called function templates.
- A function template is similar to a normal function, but only differ in one way. but, While a single normal function can only work with one set of declared data types, a single function template can work with variety of data types at same time.
- Normally, to perform identical operations on two or more types of data, we use function overloading to create two functions with the required function declaration.
- However, a better approach would be to use function templates because we can perform the same task writing less and maintainable code.
- The general form of a declaring function template is:

```
template<class T>
returntype functionname (argument of type T)
{
//body of function
//with Type T
//whenever appropriate
//……
}
```

- Thesyntax of a function template is nearly similar to that of the class template with the exception that we are defining functions and not classes. We have to use the template parameter **T** whenever necessary in the function body and the argument list.
- A swap () functiontemplate is declaredin the following example. It is supposed swap two values of a given type of data.

```
template <typename T>
void swap(T&x , T&y)
{
T temp =x;
x=y;
y=temp;
}
```

- Example 1: Following is an example of Function templates which swaps values of two numbers. Here the function has return type void.

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &arg1, T &arg2)
{
        T temp;
        temp = arg1;
        arg1 = arg2;
        arg2 = temp;
}
int main()
{
        int inum1 = 1, inum2 = 2;
        float fnum1 = 1.1, fnum2 = 2.2;
        char ch1 = 'a', ch2 = 'b';
        cout << "Before passing data to function template.\n";
        cout << "inum1 = " << inum1 << "\ninum2 = " << inum2;
        cout << "\nfnum1 = " << fnum1 << "\nfnum2 = " << fnum2;
        cout << "\nch1 = " << ch1 << "\nch2 = " << ch2;
        Swap(inum1, inum2);
        Swap(fnum1, fnum2);
```

```
        Swap(ch1, ch2);
cout << "\n\nAfter passing data to function template.\n";
        cout << "inum1 = " << inum1 << "\ninum2 = " << inum2;
        cout << "\nfnum1 = " << fnum1 << "\nfnum2 = " << fnum2;
        cout << "\nch1 = " << ch1 << "\nch2 = " << ch2;
        return 0;
}
```

Output

Before passing data to function template. inum1 = 1 inum2 = 2 fnum1 = 1.1 fnum2 = 2.2 ch1 = a ch2 = b	After passing data to function template. inum1 = 2 inum2 = 1 fnum1 = 2.2 fnum2 = 1.1 ch1 = b ch2 = a

- Explanation:
- In the above program, a function template Swap() is defined that accepts two arguments arg1 and arg2 of data type T.
- The function templateSwap() accepts two arguments and swaps them by reference.
- Since the function does not return anything the return type is void
- Example 2: Following is an example of Function templates which finds largest two values (integer, float & character). Here the function has return type T.

```
#include <iostream>
using namespace std;
// template function
template <class T>
T Greater(T arg1, T arg2)
{
        return ((arg1 > arg2) ? arg1 : arg2);
```

```
}
int main()
{
        int intval1, intval2;
        float floatval1, floatval2;
        char charval1, charval2;
        cout << "Enter two integers:\n";
        cin >> intval1 >> intval2;
        cout << Greater(intval1, intval2) <<" is Greater." << endl;
        cout << "\nEnter two floating-point numbers:\n";
        cin >> floatval1 >> floatval2;
        cout << Greater(floatval1, floatval2) <<" is Greater." << endl;
        cout << "\nEnter two characters:\n";
        cin >> charval1 >> charval2;
        cout << Greater(charval1, charval2) << " has Greater ASCII value.";
        return 0;
}
```

Output

Enter two integers: 15 12 15 is Greater.	Enter two floating-point numbers: 21.14 120.21 120.21 is Greater.	Enter two characters: m N m has Greater ASCII value.

- Explanation:
    - In the above program, a function template Greater() is defined that accepts two arguments arg1 and arg2 of data type T.
    - T signifies that argument can be of any data type.
    - Greater() function returns the Greater among the two arguments using a simple conditional operation.

11.3.3 Template arguments

- It is possible to define Function Template with Multiple Parameters of different type

- Using a comma-separated list we can use more than one generic data type in the template statement, shown below

```
template<class T1 , class T2, …..>
returntype functionname(arguments of types T1, T2, …)
{
……
}
```

- Example : Consider the following *Example with two generic types in template functions*

```
#inlude <iostream>
#include<string>
using namespace std;
template<class T1,class T2>
void display( T1 x, T2 y)
{
cout<<x<<" "<<y<<"\n";
}
```

```
int main()
{
display(2020, "JULY");
display(14.20, "Y2K");
return 0;
}
```

Output

```
2020   JULY
14.20  Y2K
```

11.3.4 Overloading of Template Functions

- It is possible to overload Template Functions just like normal functions. A template function may be overloaded in two ways
  a. by using template functions
  b. by using ordinary functions of its name.
- In case of Template Functions, the overloading resolution is accomplished as follows:
  1. Making a call to an ordinary function with an exact match.
  2. Call to a template function could be created with an exact match.
  3. The usual overloading resolution is applied to ordinary functions and the one that matches is called.

- If no match is found an error is generated. There are no automatic conversions are applied to arguments on the template functions.
- Example: Consider the following exampleto demonstrate how a template function is overloaded with an explicit function

```
#include <iostream>
#include <string>
using namespace std;
template <class T>
void output(T arg)
{
cout<<"template output:" <<arg<< "\n";
}
void output ( int arg)
{
cout<<"Explicit output: "<<arg<<"\n";
}
int main()
{
output(121);
output(121.34);
output('IDOL');
return 0;
}
```

Output

```
Explict output: 121
template output:121.34
template output: IDOL
```

11.4 STL

- STL is short for Standard Template Library.
- Itis a powerful set of C++ template classes that provide general-purpose classes and functions with templates that implement most of commonly used data structures & algorithms like vectors, lists, queues, and stacks.

- The STL provides common programming data structures and functions such as lists, arrays, stacks, etc. It is a generalized library with parameterized components.
- STL has following four core components.

1. Containers

- A container is an object that actually stores data.
- It is a way data is organized in memory.
- The STL containers are implemented by template classes & can be easily customized to hold different types of data.

2. Algorithms

- It is a procedure i.e used to process the data contained in the containers.
- STL includes many different algorithm to provide support to take such as initializing, searching,popping, sorting, merging, copying.
- They are implemented by template functions.

3. Functions

- The STL includes classes that overload the function call operator. Instances of these classes are usually known as function objects or functors.
- Functors allow the working of the associated function to be customized with the help of parameters to be passed.

4. Iterators

- It is an object like a pointer that points to an element in a container.
- We can use iterator to move through the contents of container.
- They are handles just like pointers we can increment or decrement them.
- Algorithms in STL don't work on containers, instead they work on iterators, they manipulate the data pointed by the iterators.

11.4.1. Containers

- They are of three types
- Sequence Containers
- Associative Container
- Derived Container

- **Sequence Containers**
- They store elements in a linear sequence like a line.
- Each element is related to other elements by its position along the line.
- They expand themselves to allow insertion of elements & support a variety of operations
- Following are some common containers:
- **Vector –**
  - It is a dynamic array.
  - It allows insertion & deletion & permits direct access to any element.
- **List –**
  - It is a bidirectional linear list.
  - It allows insertion & deletion anywhere.
- **Dequeue –**
  - It is a double ended queue.
  - It allows insertion & deletion at both ends.
- **Associative Container**
- Associative Containers are created to provide direct access to elements using keys.
- They are of four types.
- **Set**
  - It is an associative container for storing unique sets.
  - Here, no duplicates are allowed.
- **Multisets**
  - Duplicate are allowed.
- **Map**
  - It is an associate container for storing unique key.
  - Each key is associated with one value.

- **Multimap**
  - It is an associate container for storing key value pairs in which one key may be associated with more than one value.
  - The main difference between a map & multimap is that, a map allows only one key for a given value to be stored while multimap permits multiple key.
- **Derived Container or Container Adaptors**
- STL provides 3 derived container, stack, queue, priority queue.
- They are also known as container adaptor.
- They can be created from different sequence container.
  - **Stack** – it is a LIFO list.
  - **Queue** – it is a FIFO list.
  - **Priority queue** – it is a queue where the 1st element out is always the highest priority queue.

11.4.2 Algorithm

- A number of useful, generic algorithms are usually provided by The Standard Template Library (STL)to perform the most commonly used operations on groups/sequences of elements.
- STL provides algorithms to perform activities such as searching and sorting, each implemented to require a certain level of iterator and will work on any container that provides an interface by iterators.
- binary search is used by Searching algorithms like lower bound and binary search and usually require that the type of data must implement comparison operator < or a custom comparator function.
- Following is the list of algorithms

• Non-range algorithms	• Partitioning
• Sequential search	• Permuting elements
• Comparing ranges	• Sorting
• General iteration	• Merging
• Copying	• Heap operations
• Replacing elements	• Binary search
• Filling ranges	• Set operations
• Removing elements	• Numeric Algorithms

11.4.3 Iterator

- Iterators are normally used to point to STL containers. It is possible for an algorithm to manipulate different types of data structures/Containers using Iterators.
- The **five** different types of iterators implemented by STLare:
    1. **Input iterators** - that can only be used to read a sequence of values.
    2. **Output iterators** - that can only be used to write a sequence of values.
    3. **Forward iterators** - that can be read, written to, and move forward.
    4. **Bidirectional iterators** - these are similar to forward iterators, but they can also move back.
    5. **Random access iterators**– these can move freely or randomly any number of steps in a given operation.
- We can define bidirectional iterators to behave similar to random access iterators, since moving forward by ten steps can be done by simply moving forward one step at a time for a total of ten times.
- Distinct random-access iterators offer efficiency advantages. For example, a vector may have a random-access iterator, but a list only has a bidirectional iterator.
- Iterators are the major feature that allows the generality of the STL.
- For example, an algorithm that reverses a string can be implemented using bidirectional iterators, later this same implementation can be used for lists, vectors and deques.
- User-created containers usually have to provide an iterator that implements at least one of the available iterator interfaces, and then all the algorithms provided in the STL can be used on the container.
- This generality can prove costly at times.
- For example, when performing a search on an associative container like a map or set, it will be slower if iterators rather than by calling member functions offered by the container itself. This is because the methods ofan associative containercan take advantage of knowledge of the internal structure, which is not transparent to algorithms using iterators.

11.5 Review Question

1. Write a short note on Generic Programming
2. What are templates? Give an overview of Class Templates & Function Templates.
3. Explain the concept of Class Templates with an example
4. Explain the concept of Function Templates with an example
5. Explain with an example how Multiple Parameters of different type can be used with Templates
6. Explain with an example Overloading of Template Functions
7. Give an overview of the C++ Standard Template Library
8. Explain the components of the C++ Standard Template Library

11.6 References

Books

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, AddisonWesly
3. Absolute C++,4th Edition, Walter Savitch,Pearson Education

Web References

https://www.geeksforgeeks.org/

https://en.wikipedia.org/wiki/Generic_programming

❖❖❖

UNIT 6

12

DATABASE PROGRAMMING WITH MYSQL

Unit Structure

12.0 Objectives

- List different options to connect to databases using C++
- List the software & libraries required to demonstrate database connectivity
- Demonstrate how pre-requisites have to be correctly done
- Demonstrate database connectivity
- Demonstrate inserting and retrieving data from a database.

12.1 Introduction

- Real world applications usually require data to be stored in a database.
- This chapter demonstrates how C++ can be used to connect to the MySQL database.

- The code is quite simple but the prerequisites need to be in place and the library files need to be placed in the right location before the code can be compiled. C++ being a High-Level Language makes it easy to start coding with little understanding of underlying concepts.

12.2 Different options for database connectivity

- There are different ways to access a database in C/C++.
- All options are standard except for ODBC; its APIs are not standard.
- Almost every database vendor provides a native client library to access its database. The Client libraries are specific to the vendor; that means that the API provided will work only with the vendors database who provided it.
- Example: The client library and the API supplied by MySQL are quite different from those of PostgreSQL.
- The driver options for database programming of MySQL with C/C++ are:
    - **MySQL Client library**
        - It is implemented in the *libmysqlclient* library.
        - It is a native C API library distributed with MySQL
        - The client API library is already installed withMySQL Server installation.
    - **MySQL C/C++ Connector**
        - The API is based on the JDBC4.0 API standard
        - It provides a separate connector for C and C++.
    - **ODBC**
        - Stands for Open Database Connectivity
        - It was developed by Microsoft in the 90's.
        - it provides a vendor-neutral API to a access database system. All database vendors provide an ODBC driver other than their native support.
        - ODBC in general is a driver model, it contains the logic necessary to convert a standard set of commands into calls that can be understood by the underlying system. It can be seen as an interface between the application and the database system to facilitate the exchange of calls and responses among them.

12.3 MySQL database connectivity with C++ using C library

We will attempt the MySQL database connectivity with C++ using C library

12.3.1 Pre-requisites

We need the following software, kindly download them as per the instructions below:

1. Code Blocks
2. MySQL server (with client library)
3. Xampp

- Code Blocks
  - This will be our C++ IDE where we write our code
  - We use the version with **MinGW compiler included**
  - We have used codeblocks-20.03mingw-setup.exe(145 MB). Kindly download 32 bit or 64 bit as per your system configuration, the download link is http://www.codeblocks.org/downloads/26
- Xampp
  - Provides the integrated PHPmyAdmin interface and allows you tomanage your MySQL database
  - We have used xampp-windows-x64-7.4.8-0-VC15-installer.exe (155 MB). Kindly download 32 bit or 64 bit as per your system configuration, the download link is https://www.apachefriends.org/download.html
- MySQL server (with client library)
  - The core MySQL server
  - It is installed with the MySQL Installer, mysql-installer-web-community-8.0.21.0.exe. It installs the SQL Server along with C API. The download link ishttps://dev.mysql.com/downloads/installer/

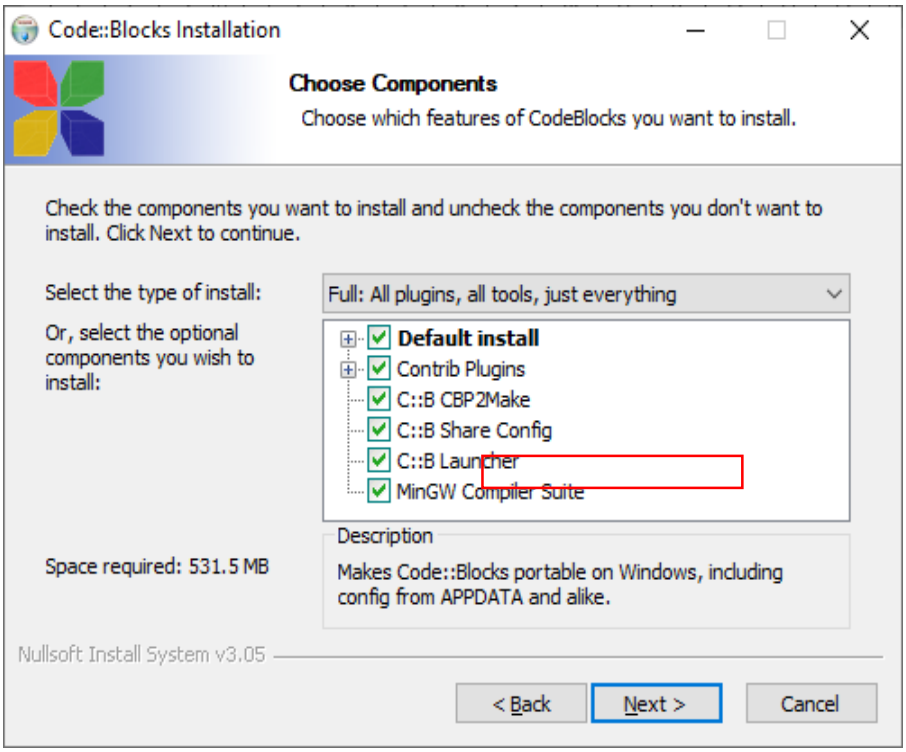
Installation and Configuration

Once the above mentioned software are downloaded install and configure them in the following order:

1. XAMPP
   - The setup is straight forward just click next wherever necessary until setup is finished

2. CODE BLOCKS

- Start the installation by double clicking the setup file codeblocks-20.03mingw-setup.exe. Proceed with the Installation. Under the **Choose Components** window make sure the MinGW Compiler Suite is present and checked

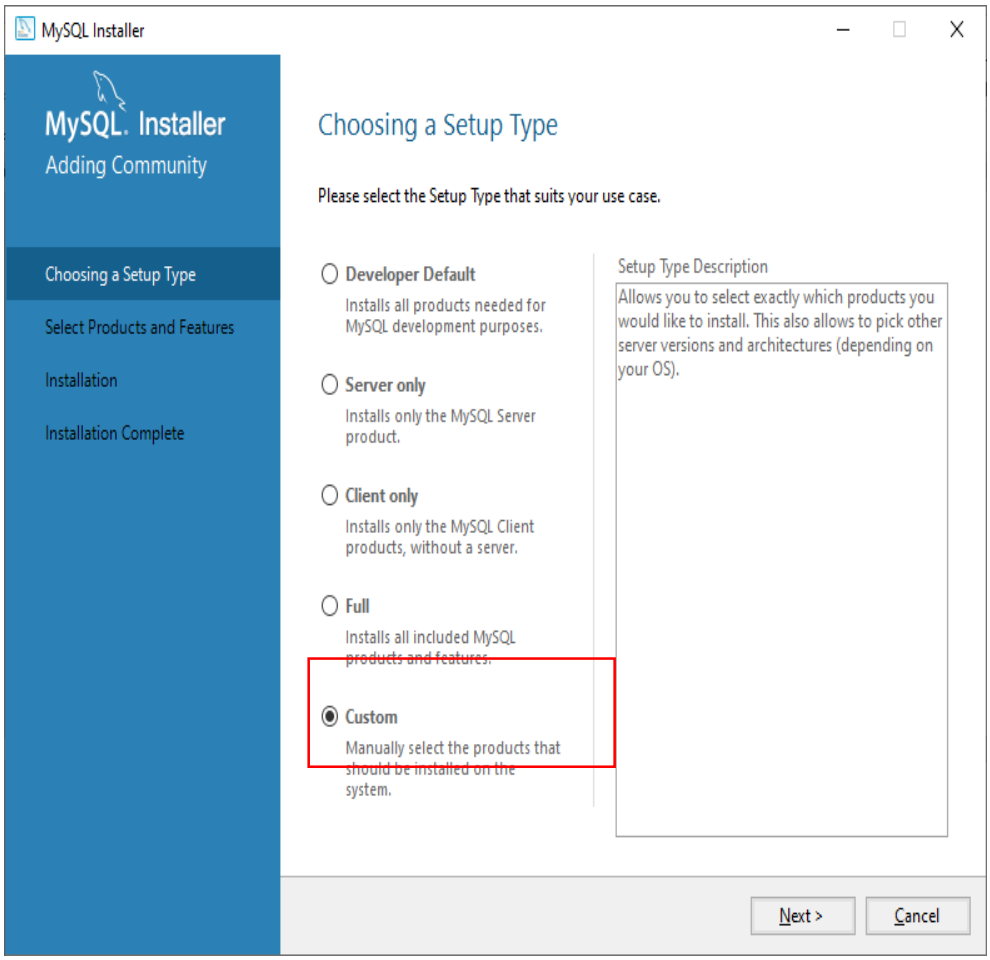


- The rest of the setup is straight forward just click next wherever necessary until setup is finished

3. MYSQL Installer

- Start the installation by double clicking the setup filemysql-installer-web-community-8.0.21.0.exe.

- Choosing a Setup Type – select custom (here we want minimum installation)

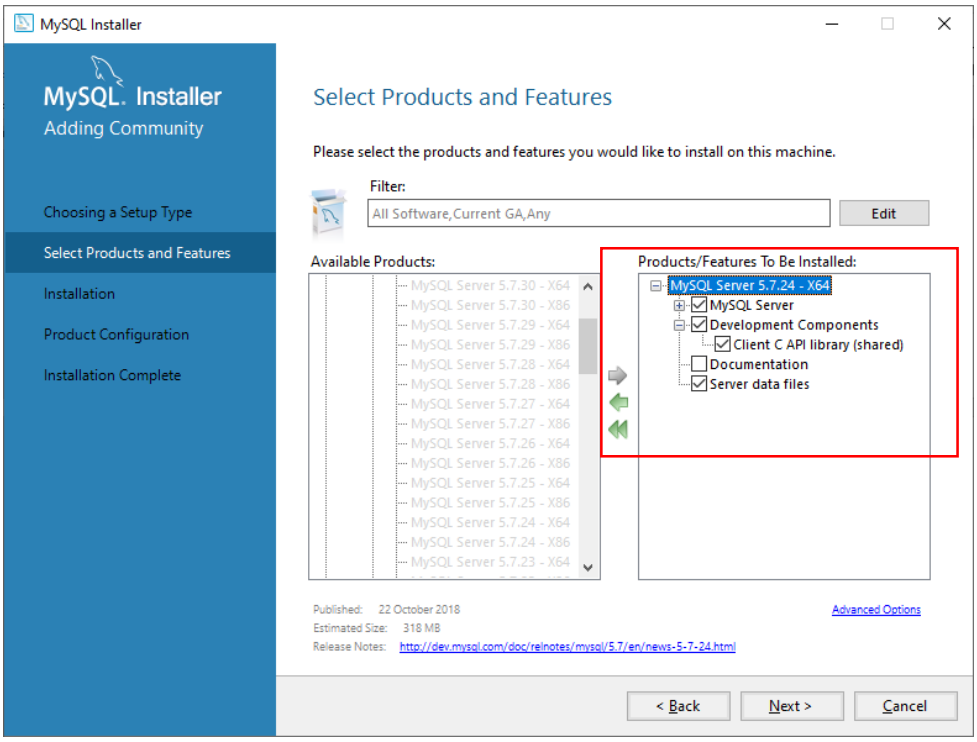


- Under Select Product & Features window do the following:

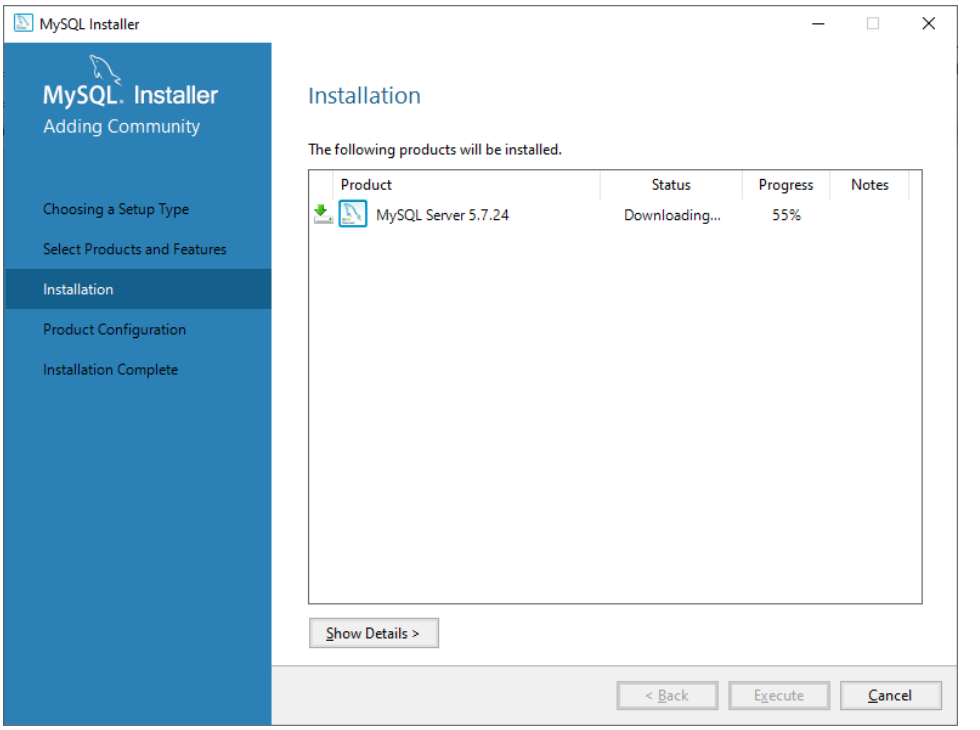
1. Select MySQL Servers
2. Expand MySQL Server
3. Expand MySQL Server 5.7
4. Select MySQL Server 5.7.24 (use x86 / x64 appropriately)
5. Click Green arrow in center importing the selection in the Products/Features to be installed panel. Make sure to expand and ensure that it includes the following:

Development Components >

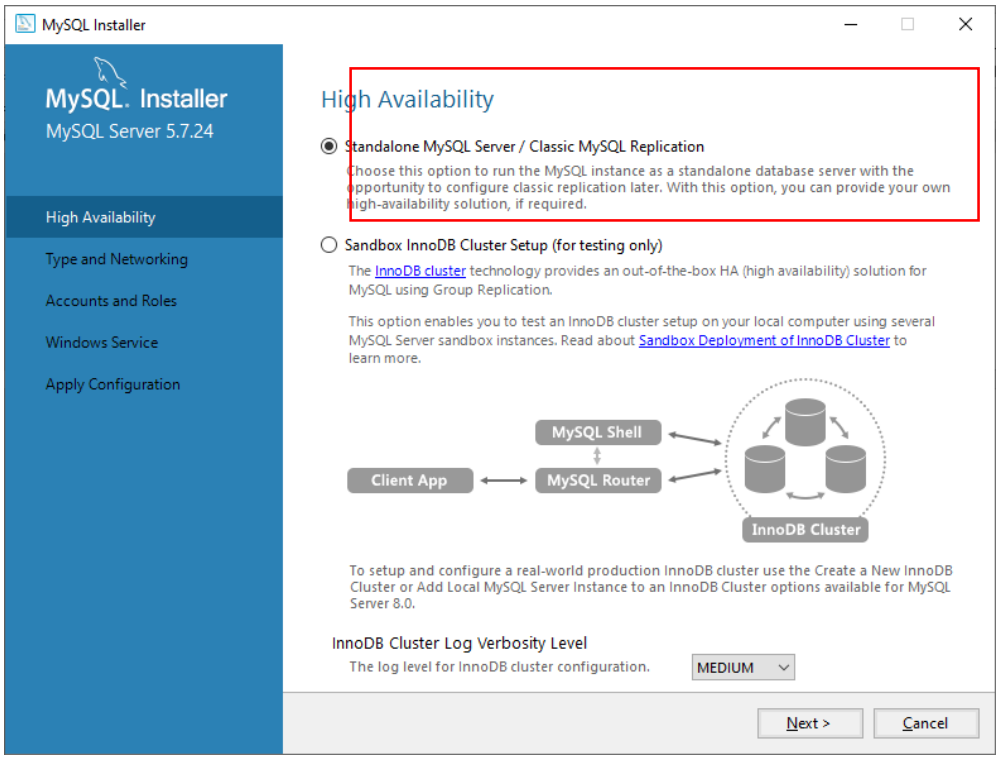
Client C API Library as shown below.



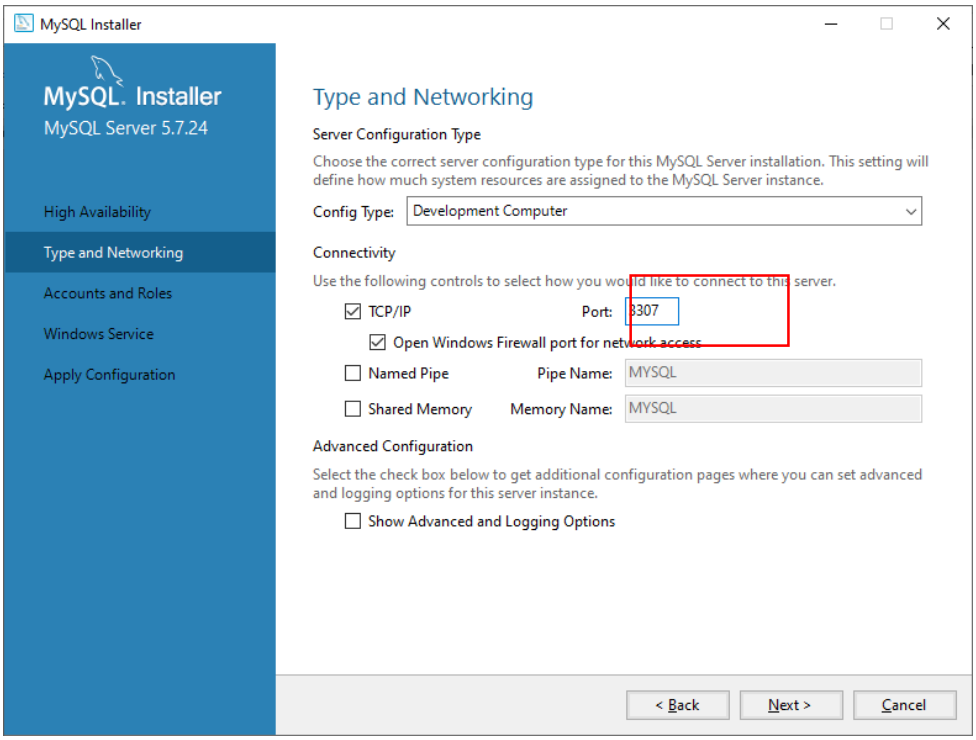
- Installation: In this window, MySQL Server 5.7.24 will be downloaded on your system (make sure internet is working)



- Product Configuration window – Click Next
- Under High Availability select Standalone MySQL Server and click Next

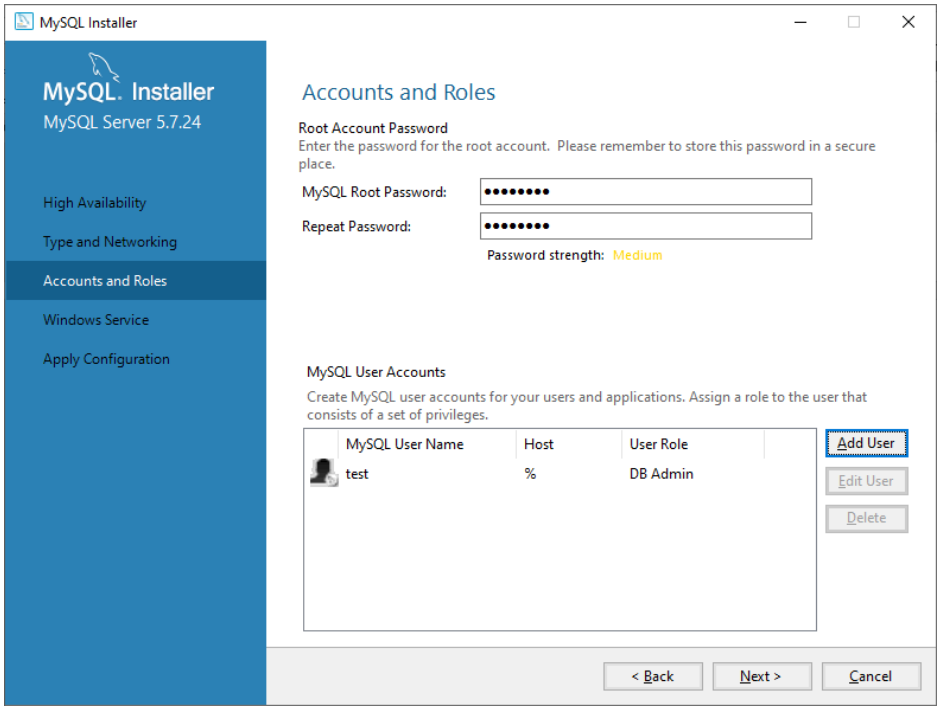


- Under Type & Networking – Change Port number 3306 to 3307 as the same is used by XAMPP and will conflict if not changed, click Next



- Accounts & Roles – set password for the default root account. Here we set it to Root@123. If you want you can add and optional user account by clicking

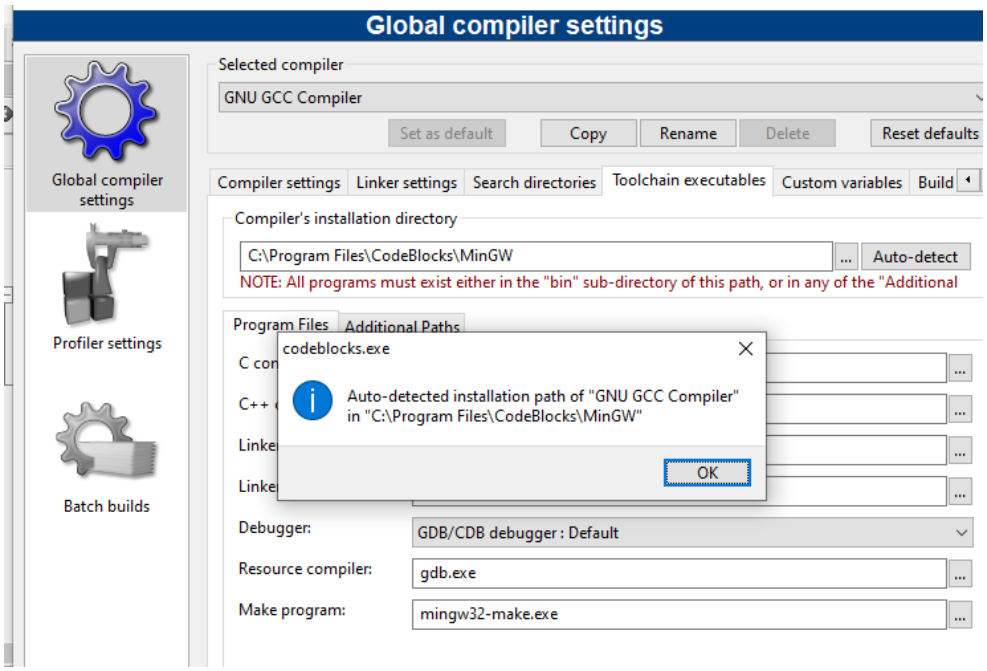
Add User. We will create a user with name “test” & password “Test@123” as shown below



- On subsequent windows click Next, Executeand Finish.
- Exit the installer

12.3.2 Setting up Code Blocks &Testing connection

- Launch Code Blocks, select Create New Project, select Console Application & Click Go, select C++ & click Next, Give a suitable title to the Project like “test” & on next window click Finish.
- Make sure you are able to successfully build & run the project.
- In case there are problems go to Settings, Compiler, Select the Toolchain Executables tab and click Auto Detect. It should automatically detect the path to GNU GCC Compiler as shown below.



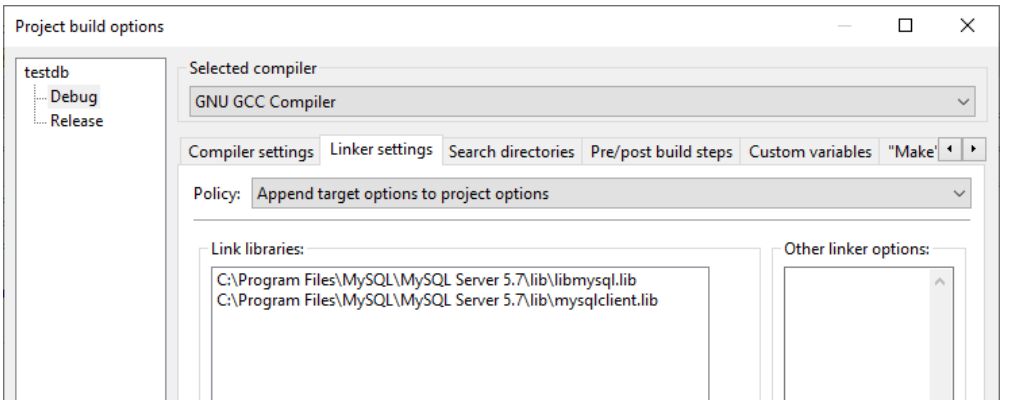
- If it does not automatically detect a path you will have to manually browse for the folder MinGW in your Code Blocks Installation which is like this C:\Program Files\CodeBlocks\MinGW

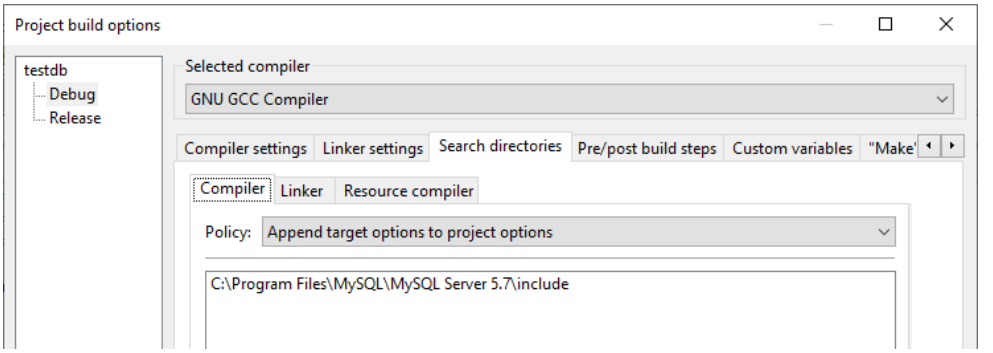
- Next add the header files <mysql.h> and try to build & run. This will give an error No such file or directory for mysql.h. To remove this error do the following:

    - Goto Project -> Build Options -> Linker Settings tab.
    - Add the following two files

C:\Program Files\MySQL\MySQL Server 5.7\lib\libmysql.lib
C:\Program Files\MySQL\MySQL Server 5.7\lib\mysqlclient.lib

    - Note: Click No if asked for Relative Path
    - Under Build Options -> Search directories tab.
    - Add the following folder -
      C:\Program Files \MySQL\MySQL Server 5.7\include
    - Note: Click No if asked for Relative Path
    - The MySQL folder may be under C:\Program Files (x86) in some systems so check and select appropriate folder.





- Next, go to the following folder "C:\Program Files \MySQL\MySQL Server 5.7\lib", locate the file "libmysql.dll". Copy and paste it in the following locations:

  1. C:\Windows\System32

  2. Your project folder where you can see *.cpp and *.cbp file

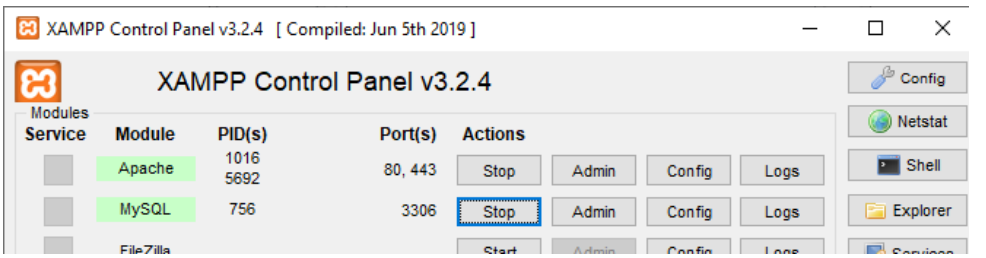
  3. Your project folders debug directory - project path>\bin\debug\

- Now, try to build & run. This time there should be no error.

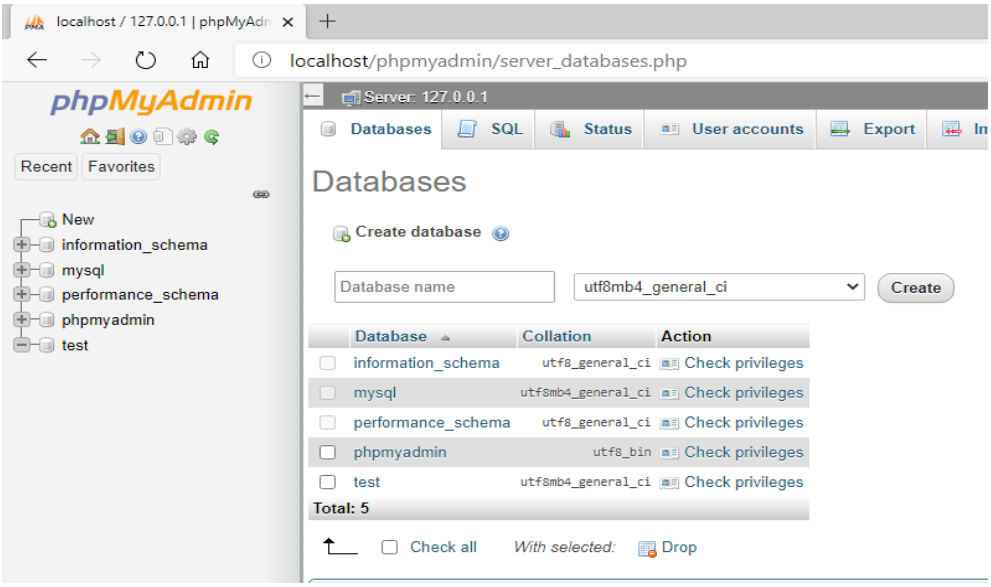
- Now let's do the actual database connectivity.

12.3.3 Connecting to MySQL database

- We start by creating a database using Xampp. Launch Xampp, then in the Xampp control panel start services for Apache and MySQL and click Admin in front of MySQL, it should look like this.



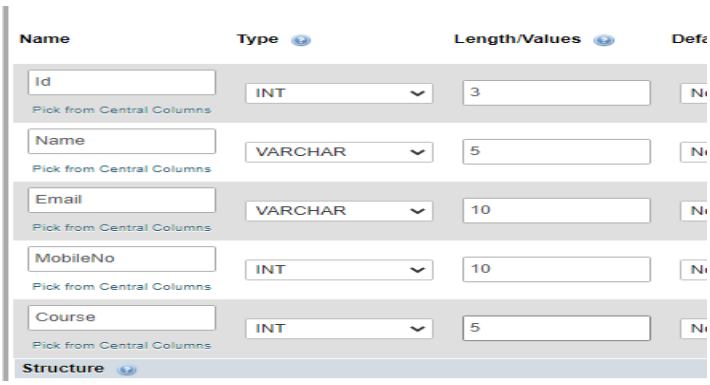
- Once you click Admin, the phpMyAdmin panel opens up in the default web browser.
- Create Database – Under the Databases Tab, type in the name of database to be created and click Create. Here we create a database called “test”



- Next, it will ask to create a table. Type the name “student” & no of columns to “5” and click “Go” to create student table
- Here we intend to create the following table

Id	Name	Email	MobileNo	Course
Int	Varchar	Varchar	Int	varchar
1	ABC	abc@gmail.com	123456	MCA
2	XYZ	Xyz@yahoo.com	258745	MSCIT
3	MNO	mno@gmail.com	789456	BSCIT
4	PQR	pqr@yahoo.com	852456	BCA

- Enter the values as shown below and click on the Save button in bottom right corner



- Return to code blocks and write the following code:

```
#include <iostream>
#include <windows.h>
#include <mysql.h>
using namespace std;
int main()
{
MYSQL* conn;                    // Create Connection Object
conn = mysql_init(0);           // Initialize Connection
conn =
mysql_real_connect(conn,"127.0.0.1","root","","test",0,NULL,0);
if(conn)
    cout<<"testdatabseconnected successfully "<<endl;
else
    cout<<"connection problem: "<<mysql_error(conn)<<endl;
cout<< "Hello world!" <<endl;
  return 0;
}
```

```
7
8   int main()
9   {
10      MYSQL* conn;
11      conn = mysql_init(0);
12
13      conn = mysql_real_connect(conn,"127.0.0.1","root","","test",0,NULL,0);
14
15      if(conn)
16          cout<<"test database connected successful "<<endl;
17      else
18          cout<<"connection problem: "<<mysql_error(conn)<<endl;
19
20
21      return 0;
22  }
23
```

```
"C:\mca slm files\code\test\bin\Debug\test.exe"
test database connected successful

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

In the above code, the following statement is used to test connection
mysql_real_connect(conn,"127.0.0.1","root","","test",0,NULL,0);

Where:

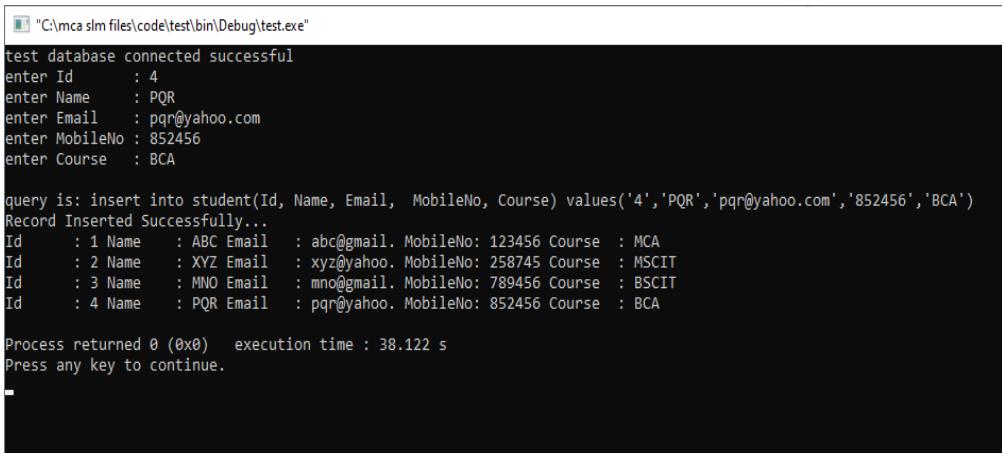
- conn = connection object,
- 127.0.0.1 = is the localhost loopback ipaddress
- root = is the username of the mysql database
- “” = no password is default for xamppphymyadmin interface
- test = name of the database to be connected

12.3.4 Inserting and retrieving data

- Next, we will add the code to insert values in the database table we created
- We will take values from user and then display the same on the screen every time a row is inserted

```
#include <iostream>
#include <windows.h>
#include <mysql.h>
#include <string>
using namespace std;
int main()
{
   MYSQL* conn;
   conn = mysql_init(0);
   conn = mysql_real_connect(conn,"127.0.0.1","root","","test",0,NULL,0);
   if(conn)
cout<<"test database connected successfully "<<endl;
   else
cout<<"connection problem: "<<mysql_error(conn)<<endl;
   MYSQL_ROW row;
   MYSQL_RES *res;
   int qstate;
   string Id, Name, Email,      MobileNo, Course;
cout<<"enter Id      : ";
cin>>Id ;
cout<<"enter Name    : ";
cin>>Name;
cout<<"enter Email   : ";
cin>>Email;
cout<<"enter MobileNo : ";
cin>>MobileNo;
cout<<"enter Course   : ";
cin>>Course;
cout<<endl;
string query="insert into student(Id, Name, Email, MobileNo, Course)
```

```
• values('"+Id+"','"+Name+"','"+Email+"','"+MobileNo+"','"+Course
+"')";
•    const char* q = query.c_str();
• cout<<"query is: "<<q<<endl;
• qstate = mysql_query(conn,q);
•    if(!qstate)
• cout<<"Record Inserted Successfully..."<<endl;
•    else
• cout<<"Query problem: "<<mysql_error(conn)<<endl;
• qstate = mysql_query(conn,"select * from student");
•       if(!qstate)
•       {
•          res = mysql_store_result(conn);
•          while(row=mysql_fetch_row(res))
•          {
• cout<<"Id      : "<<row[0]<< " "
• <<"Name    : "<<row[1]<< " "
• <<"Email   : "<<row[2]<< " "
• <<"MobileNo: "<<row[3]<< " "
• <<"Course  : "<<row[4]<< " "<<endl;
•          }
•       }
•       else
•       {
• cout<<"query error: "<<mysql_error(conn)<<endl;
•       }
• mysql_close(conn);
•    return 0;
• }
```



12.5 Review Question

1. What are the different ways of connecting MySQL database using C++?

2. What software are required for connecting MySQL database using C++?

3. List the 2 important library files needed for connecting MySQL database using C++? Also mention their path.

4. Create a database “IDOL” using phpMyAdmin interface, create a table called “MCASEM1” with columns, data types shown below. Write a program in C++ to connect to this database and Insert values as shown and display the value after every insert.

Database Name : IDOL

Table Name : COURSELIST

ID	COURSENAME	FACULTY	YEARS
Int	Varchar	Varchar	Int
1	MCA	SCIENCE	3
2	MSCIT	SCIENCE	2
3	BSCIT	SCIENCE	2
4	MSCCS	SCIENCE	2
5	BSCCS	SCIENCE	3

12.6 References

Books

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw-Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, AddisonWesly
3. Absolute C++,4th Edition, Walter Savitch,Pearson Education

Web References

https://www.geeksforgeeks.org/

https://dev.mysql.com/doc/connector-cpp/1.1/en/connector-cpp-getting-started-examples.html

❖❖❖