

UNIT-1 : Introduction to Object-Oriented Programming

Basic concept of OOP, Comparison of Procedural Programming and OOP, Benefits of OOP, C++ compilation, Abstraction, Encapsulation, Inheritance, Polymorphism, Difference between C and C++.

UNIT-2 : Elements of C++ Language

Tokens and identifiers: Character set and symbols, Keywords, C++ identifiers. Variables and constants: Integers & characters, Constants and symbolic constants, Dynamic initialization of variables, Reference variables, Basic data types in C++, Streams in C++

UNIT-3 : Operators and Manipulators

Operators, Types of Operators in C++, Precedence and Associativity, Manipulators.

UNIT-4 : Decision and Control Structures

if statement, if-else statement, switch statements, Loop: while, do-while, for; Jump statements : break, continue, go to.

UNIT-5 : Array, Pointers and Structure

Arrays, pointer, structure, unions;

UNIT-6 : Functions

main() function, components of function : prototype, function call, definition, parameter; passing arguments; types of function, inline function, function overloading.

UNIT-7 : Introduction to Classes and Objects

Classes in C++, class declaration, declaring objects, Defining Member functions, Inline member function, Array of objects, Objects as function argument, Static data member and member function, Friend function and friend class.

UNIT-8 : Constructors and Destructors

Constructors, Instantiation of objects, Default constructor, Parameterized constructor, Copy constructor and its use, Destructors, Constraints on constructors and destructors, Dynamic initialization of objects.

UNIT-9 : Operator Overloading

Overloading unary operators: Operator keyword, arguments and return value; Overloading Unary and binary operators: arithmetic operators, manipulation of strings using operators, Type conversions.

UNIT-10 : Inheritance

Derived class and base class: Defining a derived class, Accessing the base class member, Inheritance: multilevel, multiple, hierarchical, hybrid; Virtual base class, Abstract class.

UNIT-11 : Virtual Functions and Polymorphism

Virtual functions, Pure virtual functions; Polymorphism, Categorization of polymorphism techniques: Compile time polymorphism, Run time polymorphism.

UNIT-12 : File Handling

File classes, Opening and Closing a file, File modes, Manipulation of file pointers, Functions for I/O operations.

UNIT-1 : INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

UNIT STRUCTURE

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Basic Concepts of Object-Oriented Programming (OOP)
- 1.4 Comparison of Procedural Programming and OOP's
- 1.5 Benefits of Object-Oriented Programming (OOP)
- 1.6 C++ Compilation-A Quick Look
- 1.7 Some Features in OOP
 - 1.7.1 Abstraction
 - 1.7.2 Encapsulation
 - 1.7.3 Inheritance
 - 1.7.4 Polymorphism
- 1.8 Difference between C and C++
- 1.9 Let Us Sum Up
- 1.11 Answer to Check Your Progress
- 1.12 Further Readings
- 1.13 Model Questions

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to–

- acquire the concepts of OOP
- differentiate OOP with procedural language
- describe the advantages of OOP
- describe compilation process of OOP
- illustrate some basic features of OOP and PPL

1.2 INTRODUCTON

We are already familiar with the C programming language. C language is known as procedural oriented language as because C consists of sequence

of instructions i.e. the actions for the computer to follow and organizing these instructions into groups known as **function**. However, this approach always are not suitable specially for the real life applications terms of bug-free, maintainance and reusable code. To overcome this difficulty, the concept of Object-Oriented Programming (OOP) comes out.

In this unit, we will discuss the basic idea of OOP and how it is different with procedural language like C. Also we will discuss the benifits of OOP over procedural language.

1.3 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING (OOP)

As we know OOP is basically used to solve the real life applications. We can call OOP as the programming methodology that focuses on data rather than processes. It is a modular approach to computer program design. Each module or objects combines the data and procedures (sequence of instructions) that act on the data. A group of objects that have the properties, operations and behaviour in common is called a **class**.

Here in OOP's programmers define not only the data type of a data structure, but also the types of operations that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from the other objects.

May be some of you get confused with the term **Object**. Actually *object are the basic run-time entities* in an object-oriented system and every object is associated with data and functions which define meaningful operations on that object.

For object-oriented programming, we need an object-oriented programming language like Java, C++ etc. We are here going to discuss about the C++ language.

Some main features of OOP are:

- a. Encapsulation
- b. Data abstraction
- c. Inheritance
- d. Polymorphism
- e. Message passing
- f. Extensibility
- g. Persistency
- h. Data hiding

We will briefly introduce these features in this unit. An elaborate description of the above topics will be given in our next units.

Let us take an example, if **HUMAN** can be a class and Jadu, Ram etc are names of human which can be considered as object. So every human has eye, so eyecolor can be considered as the property of human being which can be encapsulated (will be discussed later) as a data in class HUMAN.

```
class HUMAN
{
    EyeColor e_color;
    NAME human_name;
}
```

Consider object of the class HUMAN is **human_obj**; we want to set human_obj's name as "Jadu" and e_color as "black". For that purposes we need a method or function.

So the required methods for the above class to do a particular task on the data are:

```
class HUMAN
{
    EyeColor e_color;
    NAME human_name;
    SetName(NAME anyName); // set the human name
    SetEColor(EyeColor color); // set the eye color
}
```

So, a *class* is a combination of *data members* (i.e *properties of object*) and *methods to manipulate that data*. But the basic thing to understand is that class has properties not any particular object informations. In other words the HUMAN class has no specific EyeColor or NAME. Only when the object of that class (e.g. human_obj) has name like "Jadu" and eye color "black".

As we have seen in fig. 1, so in real life many human may present and Jadu is one of them with the black eyes.



Fig. 1.1 : Human Class and Human Object (e.g Jadu)

1.4. Comparision of PPL and OOP

C language was created by Kernighan and Ritchie in 1970. C was created with simplicity and flexibility in mind. Its primary use was for writing operating systems (e.g. WinXP, LINUX etc). However it became popular because of its simplicity. The one thing that C lacked was support for objects; it was a procedural-oriented program.

In a procedural-based programming language, a programmer writes out instructions that are followed by a computer from start to finish. This kind of programming had its own advantages, but an OOP makes programming more clearer and easier to understand as we already discussed. Object-oriented programming is all about using objects.

An object actually contains code (member functions) and data (data mem-

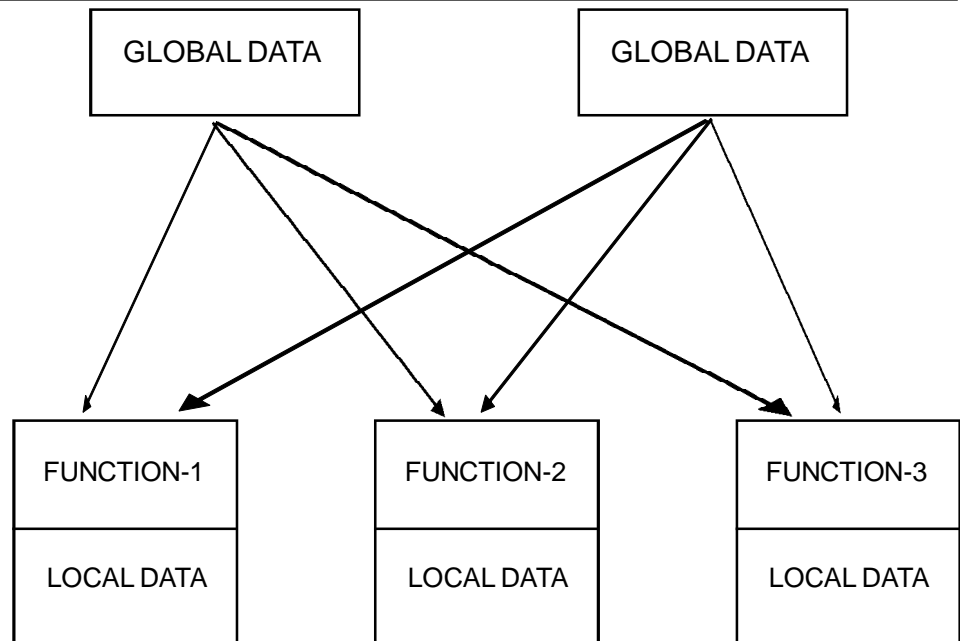
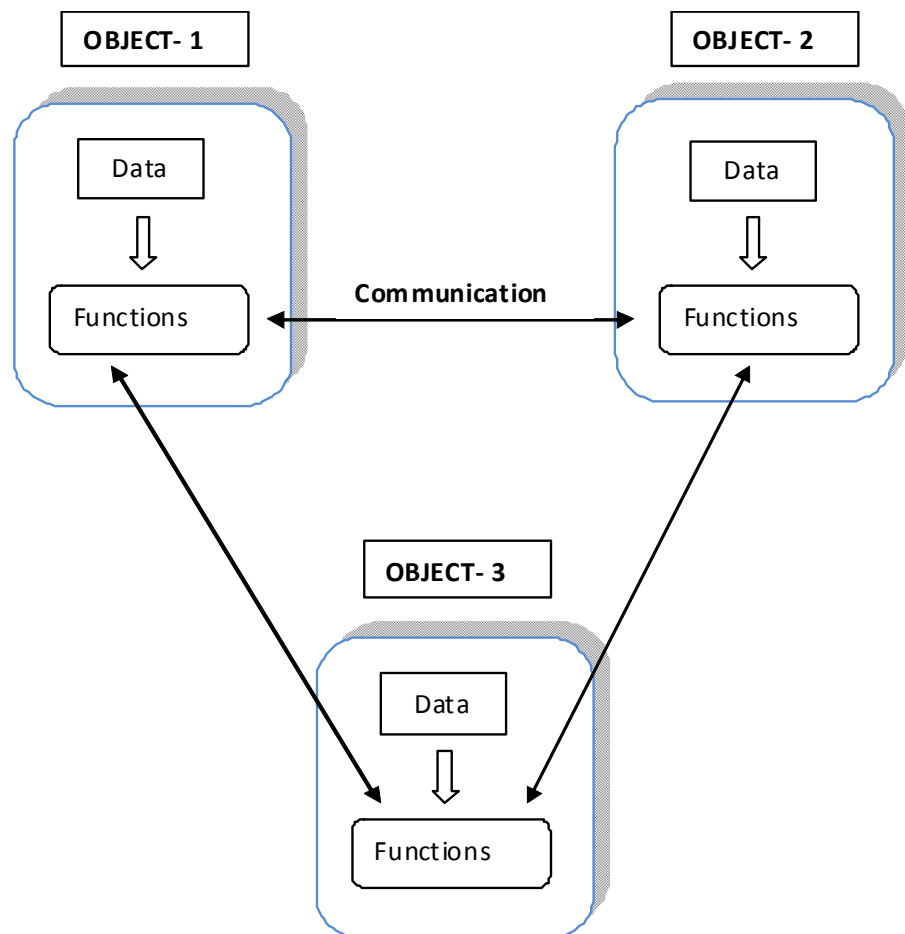
bers) where the code and data have been kept apart. For example, in the C language, units of code are called functions, while units of data are called structures. Functions and structures are not formally connected in C. A C function can operate on more than one type of structure, and more than one function can operate on the same structure.

The main difference of OOP with procedural language are:

- a) Object Orientation Languages objective is to develop an *application based on real time* while Procedural Programming Languages(PPL) are more concerned with the *processing of procedures or functions*.
- b) In OOP, more *emphasis is given on data* rather than procedures, while the programs are divided into objects and the data is encapsulated (i.e. hidden) from the external environment, providing *more security to data* which is not generally applicable or rather possible in PPL like C.
- c) In OOP, Objects communicate with each other via functions (will be discussed in next the units) while there is no explicit communication in PPL rather its simply a passing values to the arguments to the functions.
- d) OOP follows bottom up approach of program execution while in PPL its top down approach.
- e) OOP's concepts includes Inheritance, Encapsulation and Data Abstraction, Polymorphism, Multithreading, and Message Passing while PPL is simply a programming in a traditional way of calling functions and returning values.PPL does not suport it.

The list of OOP languages are :- C++, JAVA, VB.NET, C#.NET

The list of PPL languages :- C, VB, Perl, Basic, FORTRAN.

**Fig. 1.2 : Relationship of data and functions in PPL****Fig. 1.3 : Relationship of data and functions in OOP**

1.5 BENEFITS OF OOP

We have now clear idea about the OOP's and PPL and also able to identify the difference between them. As we know OOP is basically used in real time applications; it makes the applications more reliable and efficient as compared to PPL. The following are the benefits of Object-Oriented Programming over procedural programming language.

- a) Using **inheritance** concepts, we can *reuse the existing classes* i.e. program code or data structure. In other words we can eliminate the redundant code.
- b) **Data hiding** mechanism helps us to *build the secure programs* that can not be invaded by code in the other parts of the program.
- c) The **data centered design** approach helps to *acquire more details* of model in implementable form.
- d) OOP's can *easily upgrade a small system to a large system* with just a few modifications in the previous existing code or data structure.
- e) **Message passing** concepts for communication among objects *makes the interface descriptions with external system much simpler*.
- f) Using OOP concepts the *time and space complexity can be reduced*.

1.6 C++ COMPILATIONS – A QUICK LOOK

We are already familiar with C language like saving the program, compilation and execution. C++ environment is almost same with C language. Just compiler used by C++ is different than C. One important thing is that C program can be compiled and executed by the C++ compiler but the reverse is not true i.e. C++ program can not be executed / compiled by a C compiler. In other words C++ includes all the features of C languages plus some additional functionality.



Compiler :

A compiler is a computer program or set of programs that transforms source code written in a computer language (i.e. source language) into another computer language (i.e. target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

Also one important thing is that the compilations of C or C++ varies depending upon the Operating System (O.S.) used. Here we are going to discuss only about the windows operating system like Win XP, Win 98. Since we already discussed how to save, compile and run a program in C language. We are here not going for detail. Just take a quick look of it. Even If after reading the following section, if you still have confusion please go through the first section of the C language module.

We can use **Turbo C++** or **Borland C++** to write the C++ program. They are nothing but the application software for writing the C++ program. We can create and save the C++ source files under **Edit** menu. We can compile and execute the program under the **Compile** and **Run** menu respectively.

Some snapshot of **Turbo C++** is given below:-

Fig. 1.4 to Fig. 1.7 shows the simple C++ window, Save, Compile and Run menu in Turbo C++ respectively. As we see in the above pictures, the compilation of C++ program can be done by going to the compile menu. But the compilation of C++ program in **UNIX** or **LINUX** operating system are different.

Assume that **hello_world.cpp** is a C++ program written in LINUX. Then

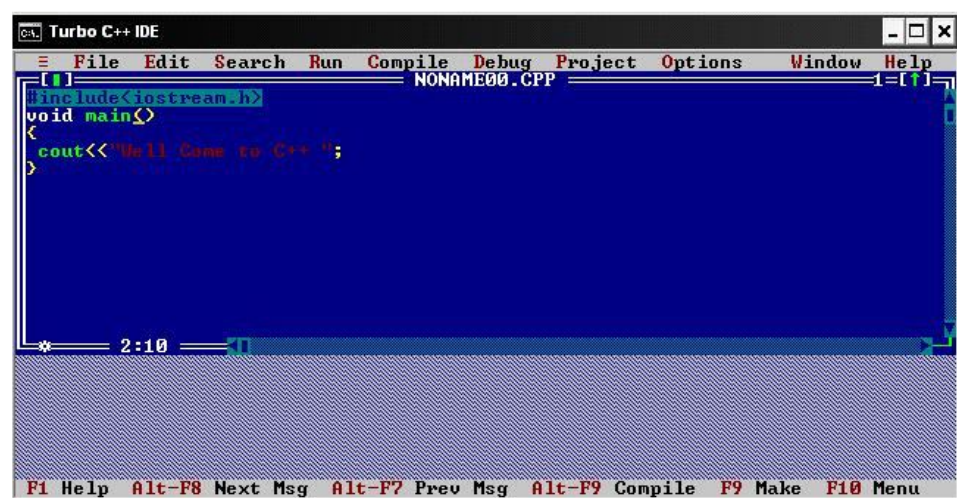


Fig. 1.4 : Simple Window of Turbo C++

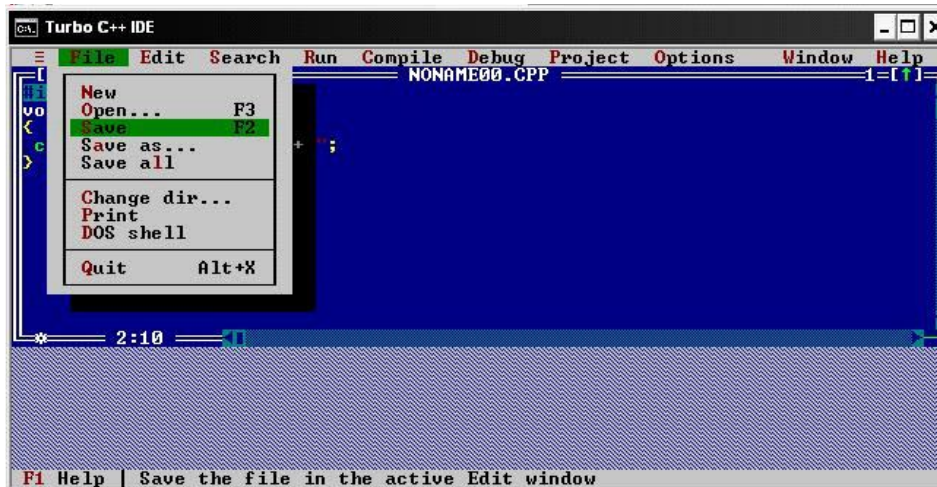


Fig. 1.5 : Save menu in Turbo C++

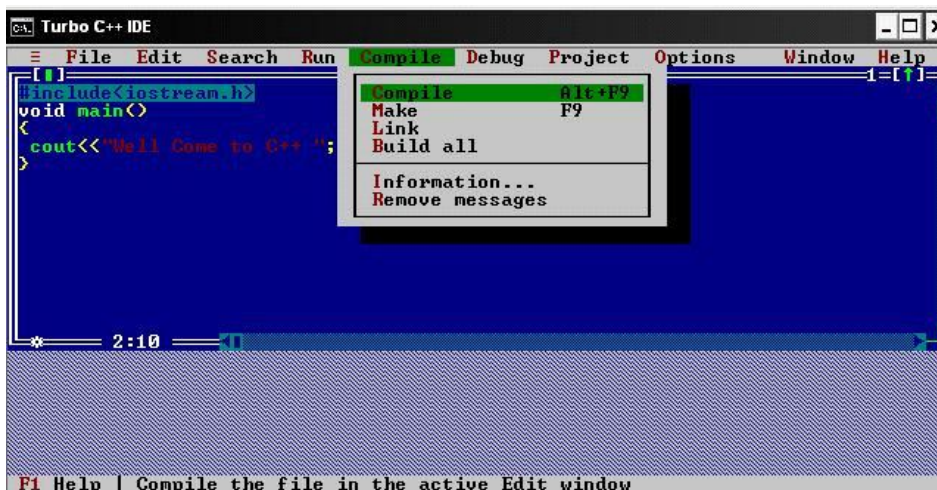


Fig. 1.6 : Compile menu in Turbo C++

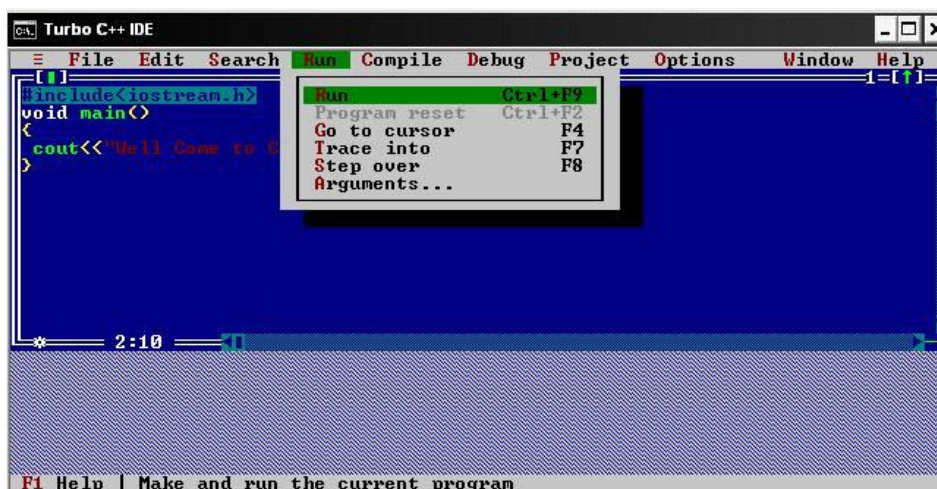


Fig. 1.7 : Run menu in Turbo C++

we need to compile it from the command prompt. It looks like:

C++ hello_world.cpp



exe : It is an executable file generated after compilation of the source file.

After the successful compilation, it will create a file **a.out**. This is nothing but an **executable** file in UNIX. To execute it we need to write in the following way:

./a.out

The main logic for any program is always same. It does not depend upon any operating system or even any compiler. So if you get confused regarding the compilation in different operating system, no need to worry, just give attention only to the logic of a program.



CHECK YOUR PROGRESS - 1

1. Determine the following whether true or false.
 - a. OOP concepts can be used to design real life applications.
 - b. C is a an Object Oriented language.
 - c. Encapsulations is special features in OOP.
 - d. Class concpets is used in C++.
2. Fill in the blanks:
 - a. _____ mechanisim helps us to build the secure programs in OOP.
 - b. In OOP, more emphasis is given on _____ rather than _____ .
 - c. OOP follows _____ approach where PPL follows _____ .
3. Give some examples of objects that you see in daily life.

.....

.....

.....

4. Write two difference between OOP and PPL.

.....
.....
.....

5. Do you think that OOP is better than PPL? Give reason.

.....
.....
.....

1.7 SOME FEATURES IN OOP

1.7.1 Abstraction

Abstraction means to show only the necessary details to the client of the object. Do you know the inner details of the monitor you are using? What happen when we switch ON monitor? Does this matter to us what is happening inside the Monitor? Obviously not ! Take another example, when we switch on our mobile phone, we are actually not concern about the internal working details (e.g circuits) of the mobile phone. Just we know how to CALL, send SMS etc. The internal functions of the mobile is hidden from the users and give us only our requirements (such as CALL, to send SMS). This concepts is called **abstraction**.

What is Abstraction ?

The concept of abstraction relates to the idea of hiding data. *The main idea behind data abstraction is to give a clear separation between properties of data type and the associated implementation details.* Thus abstraction forms the basic platform for the creation of user-defined data types called **objects**. Data abstraction is the process of refining data to its essential form. An **Abstract Data Type** is defined as a data type that is defined in terms of the operations that it supports and not in terms of its structure or implementation.

In object-oriented programming language C++, it is possible to create and provide an interface that accesses only certain elements of data types. The programmer can decide which user to give access to and hide the other details. This concept is called **data hiding** which is similar in concept to data abstraction.

Different Types of Abstraction:

There are two types of abstraction; **functional abstraction** and **data abstraction**. The main difference between them is that functional abstraction refers to a function that can be used without taking into account how the function is implemented. Data abstraction refers to the data that can be used without taking into account how the data are stored.

Why Abstraction is needed ?

There are lots of advantages for using abstraction. They can be explained as follows:

- a) Flexibility : By hiding data or abstracting details that are not needed for presentation, we can achieve greater flexibility.
 - b) Security : Abstraction gives access to only required details that are required by us and it hides the implementation details, thus giving good security to the programs. This concept helps to design real time applications with good security.
 - c) Easier Replacement: Using abstraction concepts, it is possible to replace code without recompilation. This makes our work easier and saves a lot of time.
 - d) Modular Approach: In OOP, the abstraction concept helps to divide the program into modules (small part) and test each of them individually. Then all modules are combined and ultimately tested together. This makes the application development in C++ easier.
- Hope you have now clear idea of abstraction. Here we are not going

for the implementation in detail. The next units will show the C++ program as examples.

1.7.2 Encapsulation

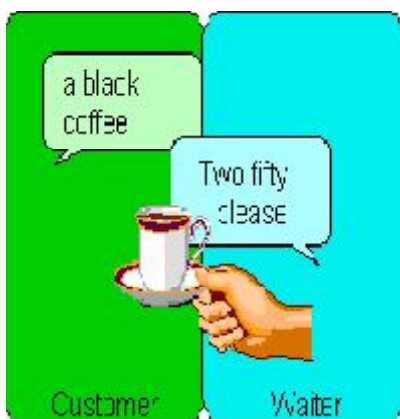
We are already familiar with the term abstraction. This abstraction concepts also brings another important idea in C++, the *encapsulation*. *Encapsulation is a method of binding the data and the codes that operates on the data into a single entity*. This keeps the data safe from outside interface and misuse.

As we know encapsulation hides the implementation details of the object and the only thing that remains externally visible is the interface of the object (i.e. the set of all messages the object can respond to). Once an object is encapsulated, its implementation details are not immediately accessible any more. Instead they are packaged and are only indirectly accessible via the interface of the object. The only way to access such an encapsulated object is via message passing-one sends a message to the object, and the object itself selects the method by which it will react to the message,determined by **functions**.

Encapsulation means as much as shielding. Take a simple example.

Say Customer, waiter and kitchen are three objects. As we know customer and kitchen do not know each other in general situation. The waiter is the intermediary between them. Objects can't see each other in an Object Oriented world. The 'hatch' enables them to communicate and exchange coffee and money.

Encapsulation keeps computer systems flexible. The business

**hatch:**

A small opening in a wall allowing access from one area to another.

process can change easily. The customer does not care about the coffee making process. Even the waiter does not care. This allows the kitchen to be reconstructed, is only the 'hatch' remains the same. It is even possible to change the entire business process. Suppose the waiter will brew coffee himself. The customer won't notice any difference even.

1.7.3 Inheritance

Inheritance is the most important feature of object-oriented programming. Inheritance is the process of creating new classes, called the derived classes, from existing or base classes. It gives a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes. The derived class inherits all the characteristics of the base class but can add some other properties of its own.

Inheritance allows us to reuse code in programming. Reusing existing code saves time and cost and hence reliability of program increases.

1.7.4 Polymorphism

Again to make clear understanding about the polymorphism, take the previous coffee example. Practically in polymorphism, objects can respond differently to the same message. Both waiter and kitchen respond to 'a black coffee' differently.



The actions are different though:

- i) The waiter passes the message to the kitchen, waits for response, delivers coffee and settles the account.
- ii) The kitchen make the fresh coffee and passes it to the waiter.

The same message with different implementations, that is the **polymorphism**. Polymorphism makes Object Oriented system extremely suitable for various exceptions used in our daily life.

So we can defined polymorphism as “the ability to use an operator or function in different ways”. *It gives different meanings or functions to the operators or functions.* The term **Poly** in polymorphism, referring to many, signifies the many uses of the operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. *Polymorphism refers to codes, operations or objects that behave differently in different contexts.*

A simple example of polymorphism is shown below :

$16+7$	$\text{"Well"}+\text{"Come"}$	$2.3+4.1$
--------	-------------------------------	-----------

As we have seen the operator + is used with different meanings with integer, strings and floating point number respectively. This concept is known as polymorphism. The above idea leads to the **operator overloading** (will be discussed in later units).

It is used to give different meanings to the same concept.

Advantages of Polymorphism:

- Helps in reusability of code.
- Provides easier maintenance of program.
- Helps in achieving robustness in the program.

Types of Polymorphism:

Mainly two types of polymorphism exists. They are:

- A. Run-time polymorphism.
- B. Compile-time polymorphism.

These will be discussed in detail in the later units.

2.8 DIFFERENCE BETWEEN C AND C++

In this section, we are going to discuss those points that makes C++ different from C. From the above explanation you may able to identify the main points where the both language differ. Almost all valid C programs are valid C++ programs, but all of the C++ features are not compatible with C. In other words we can say **C++ is a superset of C**. The main difference between C and C++ are given below:

- a) C is a procedural language whereas C++ is non-procedural language.
- b) C allows the data to flow around the functions freely whereas C++ wraps up the data and functions together, due to that data are not allowed to flow around freely in C++. (This is said to be **encapsulation** in C++).
- c) C has **no data security** while C++ has the security of data, anyone can access the data of C program while in C++ only member of the class can access the data, outside it can't access the data.
- d) In C++, it is strictly enforced that all functions must be declared before they are used. However in C we may avoid the declaration.

Take an example:

```
#include <stdio.h>

int main()
{
    msg();
    return 0;
}

void msg()
{
    printf( "Hello world" );
}
```

This works in C language but not in C++.

- e) *Type checking is much more rigid* in C++ than it is in C, so a program that compile successfully under a C compiler may result in many warnings and errors under a C++ compiler.
- f) C++ allows to declare variables and statements in any order i.e. in

any place in the program not just at the beginning but before the uses of the variable. For example, the following fragment is legal in C++ but not in C.

```
for (int j = 0; j < 10; j++)  
    s=s+j;
```

The example shows how C++ allows to declare and initialize the variable 'j' in the for loop statement instead of at the start of the function.

- g) Although it is possible to implement anything which C++ could implement in C, C++ aids to standardize a way in which objects are created and managed, whereas the C programmer who implements the same system **has a lot of liberty** on how to actually implement the internals. and **programming style among programmers will vary a lot** on the design choices they made.

The above points are the main points that differ C and C++.



CHECK YOUR PROGRESS - 2

1. Determine the following whether true or false.
 - a. Abstraction gives a clear separation between properties of data type and the associated functions.
 - b. C does not support encapsulations but support abentity called object.
 - c. Encapsulation binds the data and the functions into a single entity called object.
2. Write two advantages of abstraction.
.....
.....
.....
3. Mention two features of C++ language that are not included in

C language.

.....

1.9 LET US SUM UP

- Object oriented programming was invented to overcome the drawbacks of the procedure programming language (PPL). OOP's uses the bottom up approach while PPL uses top down.
- In OOP, a problem is considered as a collection of a number of entities called Object where the object are the basic instances of classes.
- OOP's offers several advantages over PPL. The most common one is "re-useability" of the code and data structures.
- Application of OOP technology has gained importance in almost all areas of computing including real-time system.
- Examples of OOP's are C++, JAVA, Visual Basic etc. C is procedure oriented languages.



1.10 ANSWER TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS - 1

1. a) True b) False c) True d) True
2. a) Data Hiding b) Data, procedures c) Bottom Up, Top down
3. CAR, ANIMAL etc
4. i) OOP supports polymorphism, but PPL not.
 II) OOP uses data hiding concept but PPL does not have any such concepts.
5. OOP is better than PPL since we can solve real life problem using it unlike PPL. Also OOP gives a security in accessing the data.

CHECK YOUR PROGRESS - 2

1. a) True b) False c) True
2. Flexibility and security (for detail refer to section 1.7.1)
3. C++ support polymorphism and data hiding while C does not support it.



1.11 FURTHER READINGS

1. Object Oriented Programming with C++, E. Balagurusamy, Tata McGraw Hill Publication.
2. The Complete Reference C++, Herbert Schildt, Tata McGraw Hill Publication.
3. Mastering C++, K.R. Venugopal, Rajkumar, T. Ravi Shankar, Tata McGraw Hill Publication.



1.12 MODEL QUESTIONS

1. What do you mean by Object Oriented Programming language? How it is differ with the procedure programming languages? Explain.
2. Write the basic difference between OOP and PPL.
3. Briefly explain about OOP's with an examlpe. Give 3 example of OOP languages.
4. Mention the special features in OOP's .
5. State whether the following are true or false
 - a) In PPL all data are shared by all functions in a program.
 - b) Object oriented approach can scale up better from small to large.
 - c) Objects are the basic instances of classes.
 - d) Java is a pure Object Oriented Programming language.

UNIT-2 : ELEMENTS OF C++ LANGUAGE

UNIT STRUCTURE

- 2.1 Learning Objectives
- 2.2 Introduction
- 2.3 Token, Identifier and Keywords
- 2.4 Character Set and Symbols
- 2.5 Basic Data types in C++
- 2.6 Variables
- 2.7 Constants
- 2.8 Dynamic Initialization of Variable
- 2.9 Reference Variable
- 2.10 Streams in C++
- 2.11 Let Us Sum Up
- 2.12 Answers to Check Your Progress
- 2.13 Further Readings
- 2.14 Model Questions

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about the basic elements of C++ viz. character set, token, identifier and keywords
- declare variables and constants
- describe dynamic initialization of variable and reference variables
- use the available C++ streams

2.2 INTRODUCTION

In the previous unit, we were introduced to the primary concepts that are applicable in the Object Oriented Programming paradigm. We have already come across some basic concepts of C++'s Object Oriented Programming methodology namely Abstraction, Encapsulation, Inheritance, Polymorphism etc. C++ inherits from C language an economy of

expression that novices often find cryptic. And as an Object Oriented language, its widespread use of classes and templates presents a formidable challenge to those who have not thought in those terms before. This unit provides the fundamental concepts necessary for the first-time C++ programmers. In this unit we will get to know about terms like tokens, variables, constants, data types. Streams that provide for transfer of information in the form of a sequence of bytes will also be discussed in this unit.

2.3 TOKEN IDENTIFIER AND KEYWORDS

A computer program is a sequence of elements called tokens. These tokens include keywords such as **int**, identifiers such as **main**, punctuation symbols such as **{** and operators such as **<<**. On compiling the program, the computer scans the source code, parsing it into tokens. If it finds something unexpected or doesn't find something that was expected, then it aborts the compilation and issues error messages.

A token is the smallest element of a C++ program that is meaningful to the compiler. It is a group of characters that logically belong together and the programmer can write a program by using these tokens.

Tokens are classified in the following types–

- | | | |
|-----------------------|---------------|---------------|
| (a) Keywords | (b) Variable | (c) Constants |
| (d) Special character | (e) Operators | |

Here,

- | | |
|-------------------|--|
| Keywords | – are set of reserved words with fixed meanings
e.g. <i>int, switch, char, class</i> etc. |
| Variables | – are used to hold data temporarily, eg. <i>marks, age, name</i> etc. |
| Constants | – the fixed values like 3.2, 9.3 etc. |
| Special character | – Symbols like #, ~ are known as special character |

Operators

- are used to perform different operations such as arithmetic or logic etc. e.g. +, -, :, ?, >, < etc.

IDENTIFIER :

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consists of letters and digits, in any order, except that the first character must be a letter. To construct an identifier you must obey the following points:

- o only alphabet, digit and under scores are permitted.
- o an identifiers cannot start with a digit.
- o identifiers are case sensitive, i.e. upper case and lower case letters are distinct.

In C++, there is no limit to the length of an identifier, and at least the first 1024 characters are significant. Here are some correct and incorrect identifiers name given :

<u>Correct</u>	<u>Incorrect</u>
count	1 count
names	#\$sum
tax_rate	order-no
_temp	high balance

An identifier cannot be the same as a C or C++ keyword, and should not have the same name as functions that are in the C or C++ library.

KEYWORDS :

Reserved word are the essential part of language definition. The meaning of these words has already been explained to the compiler. So, you can't use these reserved word as a variable name. All C keywords are valid in C++. There are 63 keywords in C++.

The common keywords between C and C++ are listed in Table 2.1. Table

2.2 describes the additional keywords of C++.

Table 2.1 : C & C++ common keywords.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 2.2 : Additional C++ keywords

asm	private	bool	wchar_t
catch	protected	mutable	explicit
class	public	typename	static_cast
delete	template	const_cast	export
friend	this	namespace	true
inline	throw	using	false
new	try	dynamic_cast	typeid
operator		virtual	reinterpret_cast

2.4 CHARACTER SET AND SYMBOLS

Using the valid character set and symbols a source program is created.

The following are the valid list of character set and symbols in C++.

Alphabets	A to Z, a to z and _(under score)
Digits	0 to 9
Special symbols	# , & ! ? ~ ^ { } [] () < > . : ; \$ ' " + - / * = % blank \

2.5 BASIC DATA TYPES IN C++

C++ supports a wide variety of data types. We can choose the appropriate type for writing error free programs. The data types in C++ can be classified in the following categories.

Basic Data Type	Derived Data Type	User Defined Data Types
<i>Char</i>	<i>Array</i>	<i>Structure</i>
<i>Int</i>	<i>Function</i>	<i>Union</i>
<i>Float</i>	<i>Pointer</i>	<i>Class</i>
<i>Double</i>	<i>Reference</i>	

We will concentrate on the discussion of basic data types in this unit. Derived data types such as arrays and pointers and user defined data types such as structure, union and classes are discussed in next units. Table 2.3 lists all combinations of the basic data types and modifiers along with their size and ranges—

Table 2.3 : Basic data types and size

Type	Size (Bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed sort int	4	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long float	10	3.4E-4932 to 1.1E+4932



CHECK YOUR PROGRESS

1. Fill in the blanks.

- a) A computer program is a sequence of elements called _____.
- b) The _____ separates tokens from the input stream by creating the longest token possible using the input characters in a _____ scan.
- c) In C++, the name of variables, functions, labels, and various other user-defined objects are called _____.
- d) _____ set is a set of valid characters that a language can recognize
- e) A _____ is a word that is already defined and is reserved for a unique purpose in programs written in that language.

2.6 VARIABLES

A variable is an entity that is used to represent some specified type of information within a program and whose value can be changed during the execution of the program. A variable is denoted by giving a name to it. A variable declaration associates a memory location to the variable name. It means whenever we declare a variable, a definite amount of memory location are reserved for it. The main factors associated with a variable are as follows—

- o Data type – type of the data it store i.e. char, int, float, date (user defined) etc.
- o Variable Name – the name that is given to the variable by programmer.
- o Address – address of the memory location.
- o Value – data stored in memory locations.

We have already learned in C programming, how to give names to a variable, how to declare a variable and how to initialize values to a variable. Let us

discuss briefly about it.

Variable Name :

Variable names are identifiers used to name variables. They are the symbolic names assigned to memory locations. A variable name consists of a sequence of letters and digits. The first one must be a letter. Examples of some valid variable names are—

x	sum	count
name	student_name	MAX
age	_num	dept_num

Some invalid variable names are—

2 slum	—	first character should be a letter
date birth	—	blank not allowed
Emp, record	—	, not allowed
student--age	—	illegal character (-)

Variable Declaration :

We know that a variable must be declared before using it in a program. Actually this declaration process reserves memory depending on the type of the variable. Syntax for declaring a variable is shown below -

Data type VarName1, VarName n;

The following are some valid variable declaration statements:

int x;	// x is an integer variable
int m, n, q;	// m, n, q are integer variables
float root1, root2	// root1, root2 are floating point variables

Variables can also be declared at the point of their usage as follows :

```
for ( int i = 0; i < 10; i++ )
    count << i;

int d = 10;
```

Here, variable *i* and *d* are defined at the point of their usage.

Variable Initialization:

A variable can be assigned with a value during its declaration. The assignment operator (=) is used in this case. The following syntax shows how a variable is initialize.

Data-type VariableName = constant value;

The following are the valid initialization statements :

```
int a = 20
```

```
float x = 2.25, y = 6.0925;
```

The following program demonstrates the initialization of variables.

// Program 2.1

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

```
#include<conio.h>
```

```
void main( )
```

```
{
```

```
    int x, y;        // x and y are integer type variables
```

```
    int z = 75;      // 75 is initialize to integer variables z
```

```
    float average;
```

```
    clrscr ( );
```

```
    x = z;
```

```
    y=z+50;         // value of z is add with 50 and assigns to y
```

```
    average = 5.125;
```

```
    cout <<"x=" <<x << "\n";
```

```
    cout << "y =" <<y << "\n";
```

```
    cout<<"z="<<z<< "\n";
```

```
    cout <<"average=" <<average << "\n";
```

```
    getch ( );
```

```
}
```

RUN : x = 75

y = 125

z = 75

average = 5.125

Here, the statement `cout << "x=" << x << "\n";`

displays a message 'x=' followed by the contents of the variable x and then a new line. We will discuss about the input and output operations of C++, in the next section of this unit.

2.7 CONSTANTS

The constants in C++ are applicable to the values which do not change during execution of a program. C++ has two types of constants

- literal constants
- symbolic constants

i) Literal constant :

A literal constant is just a value. For example, 10 is a literal constant. It does not have a name, just a literal value.

For example, `int x = 100`

where x is a variable of type int and 10 is a literal constant. We cannot use 10 to store another integer value and its value cannot be altered. The literal constant does not hold memory location. Depending on the type of data, literal constants are of the following types shown with examples–

<u>Example</u>	<u>Constant Type</u>
547	Integer constant
65.125	Floating point constant
0x98	Hexadecimal integer constant
0175	Octal integer constants
'a'	Character constant
"Student Name"	String constant
"1024"	String constant

Remember that a character constant is always enclosed with single quotation mark, whereas a string constant is always enclosed with a double quotation mark. Another point to remember is that an octal

integer constant is always starts with 0 and a hexadecimal integer constant with 0x.

C++ allows non graphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character. The following table gives a listing of common escape sequences.

Escape Sequence	Nongraphic Character
\a	Bell (beep)
\b	Backspace
\f	Formfeed
\n	Newline or line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

ii) Symbolic constant :

A symbolic constant is defined in the same way as variable. However, after initialization of constants the assigned value cannot be altered.

The constant can be defined in the following three ways :

- a) *# define*
 - b) The *const* keyword
 - c) The *enum* keyword
- a) The *# define* preprocessor directive can be used for defining constants as
- ```
define Maximum 100
define PI 3.142
define AGE 30
```

In the above example *Maximum*, *PI*, *AGE* symbolic constants contains the value 100, 3.142 and 30 and here it is not mentioned whether the type is *int*, *float* or *char*. Every time when the preprocessor finds the

word Maximum, PI, AGE, it will just substitute it with the values 100, 3.142 and 30 respectively.

The following program demonstrates the use of #define—

**// Program 2.2**

```
#include<iostream.h>
#include<conio.h>
#define PI 3.142
void main ()
{
 float radius, area;
 clrscr ();
 cout << "Enter the radius :";
 cin >> radius;
 area = PI * radius * radius;
 cout << "Area of the circle =" << area << "\n";
 getch ();
}
```

**RUN :**

Enter the radius : 2.5

Area of the circle = 19.6375

In the above program the statement

$$area = PI * radius * radius;$$

is translated by the preprocessor as

$$area = 3.142 * radius * radius;$$

and calculated result is stored in the variable 'area' which is displayed in the next statement.

- b) The syntax of defining variables with the *const* keyword is shown below :

***const [data type] variable name = constant value;***



The following examples illustrates the declaration of constant variable :

```
const float p1 = 3.142;
```

```
const int TRUE = 1;
```

```
const int FALSE = 0;
```

The following program demonstrates the use of constant variable and its declaration :

### // Program 2.3

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

```
#include< conio.h >
```

```
const int MAX = 5;
```

```
void main ()
```

```
{
```

```
 int i ;
```

```
 clrscr () ;
```

```
 for (i = 1; i <= MAX ; i+ +)
```

```
 cout << " The loop runs for =" << i << "times" << "\n" ;
```

```
 getch() ;
```

```
}
```

### RUN :

The loop runs for 1 times

The loop runs for 2 times

The loop runs for 3 times

The loop runs for 4 times

The loop runs for 5 times

In the above program, the *for* loop will run for 5 times because the MAX variable contains the constant value 5.

c) Constants can be defined using enumeration as given below :

Example :

```
enum { a,b,c };
```

Here a,b and c are declared as integer constants with value 0,1 and 2.

We can also assign new values to a, b and c

```
enum { a = 5, b = 10, c = 15 } ;
```

Here, a, b and c are declared as integer constants with value 5, 10 and 15.



## CHECK YOUR PROGRESS

2. State whether True or False.

- a) Variables can be used to hold different values at different times during the execution of a program.
- b) Constants are data items that change their value during the execution of the program.
- c) Decimal point or commas do not appear in any integer constant.
- d) C++ allows non graphic characters which cannot be typed directly from keyboard.
- e) Constants can be changed while the program is running.

---

## 2.8 DYNAMIC INITIALIZATION OF VARIABLE

---

The declaration and initialization of variable in a single statement at any place in the program is called as *dynamic initialization*. The dynamic initialization is always accomplished at run time i.e. when program execution is going on. Dynamic means process carried out at run time, for example, dynamic initialization, dynamic memory allocation etc.

The C++ compiler allows declaration and initialization of variables at any place in the program. In C initialization of variables can be done at the beginning of the program.

The following program illustrates the dynamic initialization of variables in C++.

**// Program 2.4**

```
#include<iostream.h>
#include<conio.h>
void main()
{
 clrscr () ;
 cout << "Enter radius : / n";
 int r ;
 cin >> r ;
 float area = 3.14 * r * r ;
 cout << "/n Area = "<< area ;
 getch () ;
}
```

**RUN :**

Enter radius = 3

Area = 28.26

In the above program variable 'r' and area are declared inside the program.

In the statement `float area = 3.14 * r * r ;`

variable `area` is declared and initialize with the value `3.14 * r * r`. This assignment is carried out at run time. Such type of declaration and initialization of a variable is called as *dynamic initialization*.

---

## 2.9 REFERENCE VARIABLE

---

C++ supports an another type of variable called reference variable. A reference variable acts as an alternative (alias) name for a previously defined variable. Recall that a variable holds only a value and we have already learn from C programming that pointer variables are used to hold the address of some other variables. A reference variable behaves similar as an ordinary variable and also as a pointer variable. Inside a program code it is used as an ordinary variable but acts as a pointer variable. The syntax for declaring a reference variable is shown below–

***Data type & Reference variable name = variable name;***

Example :

```
int sum = 100;
int & totalsum = sum;
```

Here, the variable `sum` is already declared and initialized. The second statement defines an alternative variable name i.e. `totalsum` to variable `sum`. Both the variables will display the same value, any change made to one of the variables causes change in both the variables.

The following are some examples of reference variables—

```
char ch; float m;
char & ch1 = ch; float & n = m;
```

The following program illustrates the use of reference variables.

### Program 2.5

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int x = 10, y = 11, z = 12;
 clrscr ();
 int & m = x; // variable m becomes alias of x
 cout <<"x =" <<x <<"y =" <<y <<"z =" <<z <<"m =" <<m<<"\n";
 m = y; //changes value of x to value of y
 cout<<"x="<<x<<"y="<<y<<"z="<<z<<"m="<<m<<"\n";
 m = z; // changes value of x to value of z
 cout<<"x="<<x<<"y="<<y<<"z="<<z<<"m="<<m<<"\n";
 getch ();
}
```

**RUN:**

```
x = 10 y = 11 z = 12 m = 10
x = 11 y = 11 z = 12 m = 11
x = 12 y = 11 z = 12 m = 12
```

From the above program we have seen that any change made to the

reference variable *m* also reflects in the variable *x*.

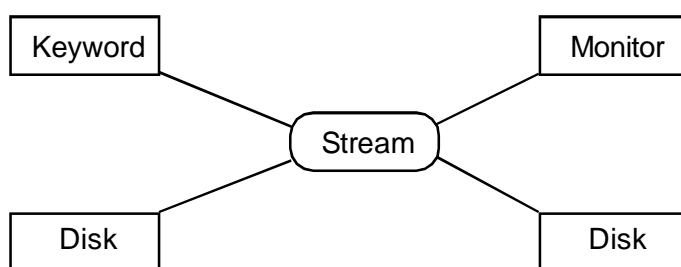
---

## 2.9 STREAMS IN C++

---

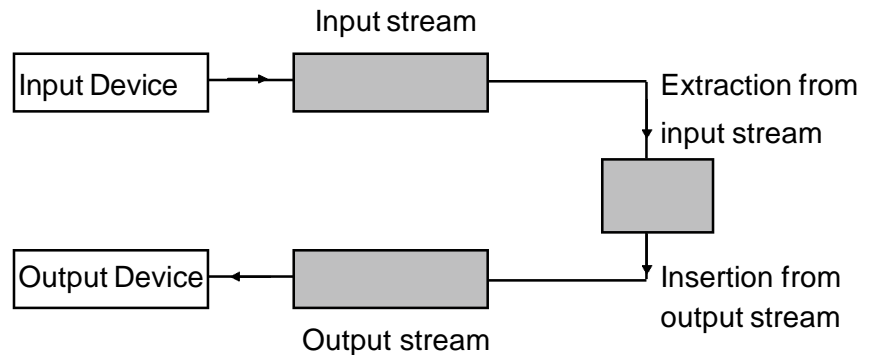
Generally every program involves in the process reading data from input device - computation is done on the data - sending the result to output devices. Hence to control such operations every language provides a set of built in functions. C++ also supports a rich set of functions for performing input and output operations. These C++ I/O functions make it possible for the user to work with different types of devices such as keyboard, monitor, disk, tape drives etc. It is designed to provide a consistent and device independent interface. These I/O functions are part of standard library. A library is nothing but a set of **.obj** (object) files.

Now we come to know that, the data flows from an input device to programs and from programs to output device. In C++, a stream is used to refer to the flow of data in bytes in sequence. If data is received from input devices in sequence then it is called as **source stream** and when the data is passed to output devices then it is called as **destination stream**. The data is received from keyboard or disk and can be passed on to monitor or to the disk. The following figure describes the concept of stream with input and output devices.



**Fig. 2.1 : Streams with I/O devices**

Data in source stream can be used as input data by program. So the source stream is also called as **input stream**. The destination stream that collects output data from the program is known as **output stream**. The mechanism of input and output stream is illustrated in the following figure.



**Fig. 2.2 : Input and output streams**

Thus, the stream acts as an intermediary or interface between I/O devices and the user. The input stream pulls the data from keyboard or storage devices such as hard disk, floppy disk etc. The data present in output stream is passed on to the output devices such as monitor, printer etc.

C++ has a number of predefined streams that are also called as standard I/O objects. These streams are automatically activated when the program execution starts. The four standard streams ***cin***, ***cout***, ***cerr*** and ***clog*** are automatically opened before the function *main()* is executed; they are closed after *main()* has completed. These predefined stream objects are declared in the header file ***iostream.h***. In this unit, we will concentrate on the discussion about *cin* and *cout*.

### Output stream :

The output stream allow to perform write operation on output devices such as monitor, disk etc. Output on the standard stream is performed using the ***cout*** object. The syntax for standard output operation is as follows :

***cout << variable;***

The *cout* object is followed by the symbol ***<<*** which is called the ***insertion operator*** and then the items (it may be variable/constants/expressions) that are to be displayed.

The following are examples of stream output operation:

`cout << "KKHSOU" ;`

```
cout << "BCA 3rd semester" ;
float area ;
cout << area ;
char code ;
cout << code ;
```

More than one item can be displayed using a single *cout* output stream object. Such output operations are called *cascaded output operations*. As an example in the above programs 2.1, 2.2, 2.4 we have already used cascaded output operation. The followings are some statements which we have used in our previous programs

```
cout << "Area =" << area ; and
cout << "x = " << x << " y = " << y << " z = " << z << " m = " << m << "\n " ;
```

The *cout* object will display all the items from left to right, we have shown in RUN portion of the program. In the first statement it will first display "Area =" and then will display the value of the 'area' variable which will be finally

Area = 28.26

The second statement will be displayed as—

x = 10   y = 11   z = 12   m = 10

Where 10,11,12,10 are the value of the variable x, y, z and m.

The complete syntax of standard output stream operation is as follows :

***cout << variable1 << variable2 <<.....<< variableN ;***

### **Input stream :**

The input stream allows to perform read operation through input devices such as keyboard, disk etc. Input from the standard stream is performed using the ***cin*** object. The syntax for standard input operation is as follows :

***cin >> variable ;***

The *cin* object is followed by the symbol ***>>*** which is called the ***extraction operator*** and then the variable, into which the input data is to be stored.

The following are some example of standard input operations :

```

int r ;
cin >> r ;
float radius ;
cin>>radius;
char name [25] ;
cin >> name ;

```

Using the *cin* input stream object inputting of more than one item can also be performed. The complete syntax of the standard input stream operation is as follows:

***cin >> variable 1 >> variable 2 >>.....>>variable N;***

Following are some valid input statements;

```

cin >> i >> j >> k ;
cin >> name >> age >> address ;

```

The following program illustrates the use of *cin* and *cout* object :

### **// Program 2.6**

```

#include<iostream.h>
#include<conio.h>
void main ()
{
 int marks1, marks2, marks3 ;
 char name [25] ;
 char semester [15] ;
 clrscr () ;
 cout << "=====";
 cout << "\n " ;
 cout << " Enter Marks : " ;
 cin >> marks1 >> marks2 >> marks3 ;
 cout << " Enter Name : " ;
 cin >> name ;
 cout << "Enter Semester : ;
 cin >> semester ; << "/n";
 cout << Marks 1 = "<<marks1<<"\n"<< "Marks 2 = "marks2 <<"\n"
 <<"Marks 3 = " <<marks 3 << "\n" ;

```

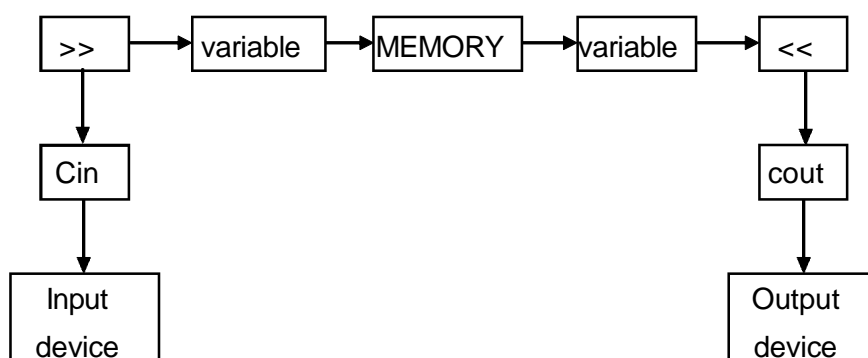


```
cout << "/n =====The End =====" ;
getch () ;
}
```

**RUN :**

```
=====
Enter Marks : 61 71 59
Enter name : Bikash Bora
Enter Semester : 3rd Semester
Marks 1 = 61
Marks 2 = 71
Marks 3 = 59
=====The End =====
```

The following figure shows flow of input and output stream :



**Fig. 2.3 : Working of *cin* and *cout* statement**



### CHECK YOUR PROCESS

3. State whether the following statements are TRUE or FALSE
  - (a) A variable name can consists of letters, digits and underscore (-) but no other special characters.
  - (b) In dynamic initialization, we initialize a variable at compile time.
  - (c) In C++, an identifier must be initialized using constant expressions.

(d) \$age is a valid variable name.

(e) Cin is also called extraction operator.

4. Choose the correct answer from the following:

(i) Which of the following is a reserved word in C++:

(a) template

(b) throw

(c) this

(d) all of the above

(ii) A variable defined within a block is visible :

(a) within a block

(b) within a function

(c) both (a) and (b)

(d) none of the above

(iii) The cin and cout functions require the header file to include:

(a) isotream.h

(b) stdio.h

(c) iomanip.h

(d) none of the above

(iv) The streams is a :

(a) flow of data

(b) flow of integers

(c) flow of statements

(d) none of the above

(v) Which of the following is C++ standard stream :

(a) cin

(b) cout

(c) cerr

(d) All of the above

---

## 2.10 LET US SUM UP

---

1. In C++, tokens are the various elements present in a program. Tokens can be classified as - keyword, variable, constants, special character and operators.
2. Identifiers are names of variables, function and arrays. They are user-defined names, consisting of sequence of letters and digits, with a letter as a first character,
3. The C++ keywords are reserved words by the compiler. All C language keywords are valid in C++ and few additional keywords are added.
4. Variables are used to store value i.e. information. A variable is a sequence of memory locations, which are used to store assigned

values.

5. C++ permits declaration of variables anywhere in the program.
6. The constants in C++ are applicable to those values, which do not change during execution of a program. The two types of constants are *literal and symbolic*.
7. The initialization of variable at run-time is called as *dynamic initialization*.
8. In C++, a reference variable acts as an alternative (alias) name for a perviously defined variable.
9. C++ supports all data type in C.
10. A stream is a series of bytes that acts as a source and destination for data. The source stream is called input stream and the destination stream is called output stream.
11. The *cin*, *cout*, *cerr* and *clog* are predefined streams.
12. The header file *iostream.h* must be include when we use *cin* and *cout* functions.



## 2.11 ANSWER TO CHECK YOUR PROGRESS

---

### Check Your Progress

1. a) Tokens. b) parser, left-to-right, c) identifiers, d) character e) keyword.
2. a)True, b)False, c)True, d)True, e)False
- 3.(a) T,                      (b) F,                      (c) F,                      (d) F,                      (e) T.
- 4.(i) d,                      (ii) a,                      (iii) a,                      (iv) a,                      (v) d.



## 2.12 FURTHER READING

---

1. Object Oriented Programming with C++, E. Bala gurusamy, Tata-McGraw Hill Publication.
2. The Complete Reference C++, Herbert Schildt, Tata-McGraw Hill Publication.
3. Mastering C++, K.R. Venugopal, Rajkumar, T. Ravi Shankar, Tata Mc-Graw Hill Publication.



---

## 2.13 MODEL QUESTIONS

---

1. What are identifiers, variables and constants?
2. What is the difference between a keyword and an identifier?
3. List the rules of naming an identifier in C++?
4. Which are the two types of constants? Describe them with suitable examples ?
5. What is dynamic initialization? Is it possible in C?
6. What are the difference between reference variables and normal variables?
7. Write short note on the following:
  - (a) Dynamic initialization of variable
  - (b) Reference variable
  - (c) Input stream
  - (d) Output stream
  - (e) Constants
  - (f) Variables
1. What are C++ Tokens and Identifiers? Give examples.
2. Give a list of some C++ Symbols and their usage.
3. What are C++ keywords? Why are they used for?
4. What are variables and constants in C++? Give examples.
5. Describe character constants and symbolic constants with examples.
6. How will you dynamically initialize a variable in C++?
7. Show with an example the use of reference variables.
8. What are C++ streams? Give examples.

## UNIT - 3: OPERATORS AND MANIPULATORS

### UNIT STRUCTURE

- 3.1 Learning Objectives
- 3.2 Introduction
- 3.3 Operators
- 3.4 Types of operators in C++
- 3.5 Precedence and Associativity of Operators
- 3.6 Manipulator
- 3.7 Let Us Sum Up
- 3.8 Answers To Check Your Progress
- 3.9 Further Readings
- 3.10 Model Questions

---

### 3.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to:

- know what are C++ operator
- use the different C++ operators
- know what is precedence and associativity of operators
- learn what are manipulators
- use manipulators in C++ programs

---

### 3.2 INTRODUCTION

---

In the previous units we had a beginning introduction to the C++ programming language. In particular, the basic concepts of object oriented programming's that are fundamental and the all important aspect of the C++ language were dealt with in brief. Also the elements of the C++ programming language like identifiers, keywords, variables etc were introduced.

Operators play a vital role in all programming languages. Using these operators we can easily use and manipulate a C++ entity to get the desired output. In this unit we will get to know what operators are in the C++ programming language. Precedence and

Associativity of operators that enable a C++ programmer to use all its language features will also be discussed in this unit.

---

### 3.3 OPERATORS

---

C++ is very rich in built-in operators. In fact it places more significance on operators than do most other computer languages. C++ Operators are special symbols used for specific purposes. Once we know about the existence of variables and constants, we can begin to operate with them using the various types of operators that are present in the C++ language. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes the code used in the C++ language shorter, since it relies less on English words, but requires a little of learning effort in the beginning.

---

### 3.4 TYPES OF OPERATORS IN C++

---

C++ provides several types of operators:

- Arithmetic operators
- Relational operators
- Logical operators
- Unary operators
- Assignment operators
- Bitwise operators
- Conditional operators
- Comma operator
- Explicit type casting operator
- The sizeof operator
- Scope resolution operator
- Insertion and extraction operator
- Address and Indirection operator
- Memory Management operator

#### 1. Arithmetic operators:

These operators perform arithmetic (numeric) operations: +, -, \*, / , or %. For these operations always two or more than two operands are required. Therefore

these operators are called binary operators. The following table shows the arithmetic operators.

| Operator name  |        | Syntax             |
|----------------|--------|--------------------|
| Assignment     |        | <code>a = b</code> |
| Addition       |        | <code>a + b</code> |
| Subtraction    |        | <code>a - b</code> |
| Unary plus     |        | <code>+a</code>    |
| Unary minus    |        | <code>-a</code>    |
| Multiplication |        | <code>a * b</code> |
| Division       |        | <code>a / b</code> |
| Modulo         |        | <code>a % b</code> |
| Increment      | Prefix | <code>++a</code>   |
|                | Suffix | <code>a++</code>   |
| Decrement      | Prefix | <code>--a</code>   |
|                | Suffix | <code>a--</code>   |

## 2. Relational/Comparison operators:

These operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

| Operator name            | Syntax                 |
|--------------------------|------------------------|
| Equal to                 | <code>a == b</code>    |
| Not equal to             | <code>a != b</code>    |
| Greater than             | <code>a &gt; b</code>  |
| Less than                | <code>a &lt; b</code>  |
| Greater than or equal to | <code>a &gt;= b</code> |
| Less than or equal to    | <code>a &lt;= b</code> |

## 3. Logical operators:

The logical operators are used to combine one or more relational expression. The following table shows the logical operators.

| Operator name          | Syntax                      |
|------------------------|-----------------------------|
| Logical negation (NOT) | <code>!a</code>             |
| Logical AND            | <code>a &amp;&amp; b</code> |
| Logical OR             | <code>a    b</code>         |

## 4. Unary operators:

C++ provides two unary operators for which only one variable is required. The following table shows the logical operators.

| Operator name | Syntax |
|---------------|--------|
| Unary plus    | +      |
| Unary minus   | -      |

**Example:**

```
a = - 50; a = - b;
a = + 50; a = + b;
```

Here plus sign (+) and minus sign (-) are unary because they are not used between two variables.

### 5. Assignment operators:

The assignment operator '=' stores the value of the expression on the right hand side of the equal sign to the operand on the left hand side. In addition to standard assignment operator shown above, C++ also support compound assignment operators.

| Operator name                  | Syntax |
|--------------------------------|--------|
| Addition assignment            | a += b |
| Subtraction assignment         | a -= b |
| Multiplication assignment      | a *= b |
| Division assignment            | a /= b |
| Modulo assignment              | a %= b |
| Bitwise AND assignment         | a&= b  |
| Bitwise OR assignment          | a  = b |
| Bitwise XOR assignment         | a ^= b |
| Bitwise left shift assignment  | a<<= b |
| Bitwise right shift assignment | a>>= b |

### 6. Bitwise operators:

Bitwise operators modify variables considering the bit patterns that represent the values they store. The following table shows the logical operators.

| Operator name | Syntax |
|---------------|--------|
| Bitwise NOT   | ~a     |
| Bitwise AND   | a& b   |
| Bitwise OR    | a   b  |
| Bitwise XOR   | a ^ b  |



|                     |                          |
|---------------------|--------------------------|
| Bitwise left shift  | <code>a&lt;&lt; b</code> |
| Bitwise right shift | <code>a&gt;&gt; b</code> |

### 7. Conditional operators:

The conditional operator `?:` is called ternary operator as it requires three operands. The format of the conditional operator is:

`Conditional_expression ? expression1 : expression2;`

If the value of `conditional_expression` is true then the `expression1` is evaluated, otherwise `expression2` is evaluated.

```
int a = 5;
int b = 6;
big = (a > b) ? a : b;
```

The condition evaluates to false, therefore `big` gets the value from `b` and it becomes 6.

**Example:**

```
#include<iostream.h>
#include<conio.h>
int main()
{
 int age;
 clrscr();
 cout<<"Enter your age in years: ";
 cin>>age;
 (age>=18)?(cout<<"\nCan vote\n"):(cout<<"Cannot
 vote");
 getch();
 return 0;
}
```

If we enter age as 26, the output is:

```
Enter your age in years: 26
Can vote
```

Again if we run the program by entering age as 15, the output is:

```
Enter your age in years:15
Cannot vote
```

**8. Comma operator:**

The comma operator gives left to right evaluation of expressions. It enables to put more than one expression separated by comma on a single line.

**Example**

```
int i = 20, j = 25;
intsq = i * i, cube = j * j * j;
```

In the above statements, comma is used as a separator between two statements/expressions.

**9. Explicit type casting operator:**

Type casting operators allow us to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses ():

```
int i;
float f = 3.14;
```

```
i = (int)f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the type casting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int(f);
```

Both ways of type casting are valid in C++.

#### 10. The sizeof operator:

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof(char);
```

This will assign the value 1 to **a** because **char** is a one-byte long type. The value returned by **sizeof** is a constant, so it is always determined before program execution.

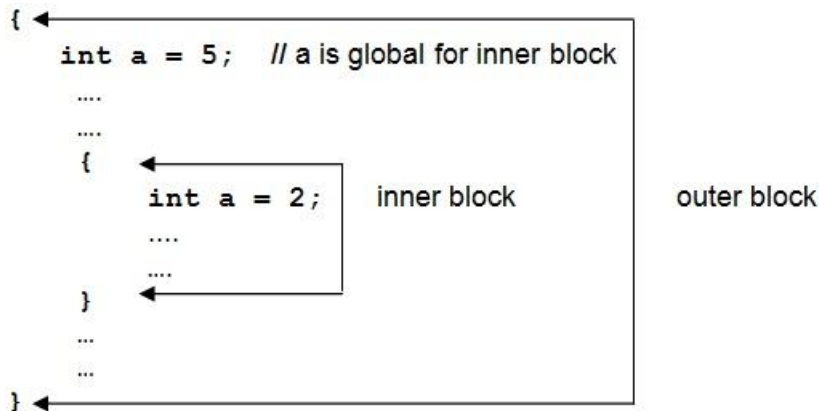
#### Example:

```
double a;
char c;
cout<<sizeof(c); //returns 1
cout<<sizeof(int); //returns 2
cout<<sizeof(a); //returns 8
```

will output 1, 2, and 8 as the size of character, integer and double are 1, 2 and 8 bytes respectively.

#### 11. Scope resolution operator:

We can use nested blocks in C++. For example, we can write nested blocks as follows:



Declaration of variable in an inner block hides a declaration of the same variable in an outer block. A variable declared inside a block is said to be local to that block. In C, a global version of a variable cannot be accessed from within the inner block. But in C++, we can resolve this problem using a new operator:: called the scope resolution operator. It can be written as:

::variable\_name;

### Example:

```

#include<iostream.h>
#include<conio.h>
int a=20; // global a
int main()
{
 int a=5; // a is local to main()
 clrscr();
 {
 int a=15; // a is local to inner block
 cout<<"Inner block a is = "<<a; // a is 15
 }
 cout<<"In outer block a is = "<<a; // a is 5
 cout<<"Outside main() a is = "<<::a; // a is 20
 getch();
 return 0;
}

```

### Output:

```

Inner block a is = 15
In outer block a is = 5
Outside main() a is = 20

```

We have used ::a to display the value of the global variable, in case we use a the output shall be 5 instead of 20. One major

application of the scope resolution operator is to identify the class to which a member function belongs.

## 12. Insertion and Extraction operator:

**Output:** The Insertion operator

To get information out of a file or a program, we need to explicitly instruct the computer to output the desired information.

One way of accomplishing this in C++ is with the use of an output stream. In order to use the standard I/O streams, we must have in our program the pre-compiler directive:

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

In order to do output to the screen, we merely use a statement like:

```
cout<< " X = " << X;
```

where X is the name of some variable or constant that we want to write to the screen.

**Input:** The Extraction operator

To get information into a file or a program, we need to explicitly instruct the computer to acquire the desired information.

One way of accomplishing this in C++ is with the use of an input stream. As with the standard input stream, cin, the program must use the pre-compiler directive:

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

In order to do output, we merely use a statement like:

```
cin>>X;
```

where X is the name of some variable that we want to store the value that will be read from the keyboard. As with the insertion operator, extractions from an input stream can also be "chained". The left-most side must be the name of an input stream variable.

## 13. Address and Indirection operator:

Program variables are allocated space in computer memory and therefore they have

individual addresses for their specific memory locations. Such an address of a variable in a program can be easily accessed by using the address operator (&). This operator when used as a prefix to a variable name returns the address of that variable.

The indirection operator (\*) returns the value of the variable located at the address that follows it.

**Example:**

```
#include<iostream.h>
#include<conio.h>
int main()
{
 int a, *p;
 a=50;
 p=&a;
 clrscr();
 cout<<"Value of a: "<<a<<endl;
 cout<<"Value of a: "<<*p<<endl;
 cout<<"Value of a: "<<*(&a)<<endl;
 cout<<"Address of a: "<<&a<<endl;
 cout<<"Address of a: "<<p;
 getch();
 return 0;
}
```

**Output:****14. Memory management operator:**

We have already come across some memory allocation and de-allocation functions in the C language. Apart from those functions, C++ also defines two operators for allocation and de-allocation of memory in an easier way. These two operators are new and delete.

**new operator:**

The new operator allocates memory of specified type and returns back the starting address to the pointer variable. The general form of new operator is:

`pointer_variable = new data_type[size];`

Here, the pointer\_variable is a pointer to the data\_type. The size is optional. We can specify the size when we want to allocate memory space for user defined data types such as arrays, structure and classes. If the new operator fails to allocate memory, it returns NULL. For example, let us consider the following declaration:

```
int *p ;
p = new int ;
char *q = new char ;
```

where p is the pointer of type int and q is a pointer of type char. For allocation of memory for user defined data type such as array, we can use the following form:

`pointer_variable = new data_type[size] ;`

The statement `int *p = new int[10];` creates memory space for an array of 10 integers (i.e., 20 bytes). `p[0]` will refer to the first element, `p[1]` to the second element and so on.

We can also initialize the memory using the new operator like this:

`pointer_variable = new data_type(value);`

For example, `int *ptr = new int(5);` where 5 is assigned to pointer variable ptr.

### **delete operator:**

The delete operator releases the memory allocated by the new operator. Following are the syntax of delete operator:

```
delete pointer_variable;
```

delete [size] pointer\_variable;

For example, delete p;  
delete [10] p;



## CHECK YOUR PROGRESS

1. Fill in the blanks.

- a) Arithmetic operators are called \_\_\_\_\_ operators.
- b) A relational expression returns zero when the relation is \_\_\_\_\_ and a non-zero when it is \_\_\_\_\_.
- c) The \_\_\_\_\_ operators are used to combine one or more relational expression.
- d) C++ provides two \_\_\_\_\_ operators for which only one variable is required.
- e) The \_\_\_\_\_ operator '=' stores the value of the expression on the \_\_\_\_\_ hand side of the equal sign to the operand on the \_\_\_\_\_ hand side.
- f) \_\_\_\_\_ operators modify variables considering the bit patterns that represent the values they store.
- g) The conditional operator '?:' is called \_\_\_\_\_ operator as it requires three operands.
- h) The \_\_\_\_\_ operator accepts one parameter.
- i) The \_\_\_\_\_ operator when used as a prefix to a variable name returns the \_\_\_\_\_ of that variable.
- j) The \_\_\_\_\_ operator returns the value of the variable located at the address that follows it.

## 3.5 PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

The precedence rules of a language specify which operator is evaluated first when two operators with different precedence are adjacent in an expression. Adjacent operators are separated by a single operand. The C++ language includes all C operators and



adds several new operators. Operators specify an evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)
- Three operands (ternary operator)

Operators follow a strict precedence, which defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right; this is called "associativity." When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

$$a = 5 + 7 \% 2$$

we may doubt if it really means:

$$a = 5 + (7 \% 2) \quad // \text{ with a result of 6, or}$$
$$a = (5 + 7) \% 2 \quad // \text{ with a result of 0}$$

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++.

The following is a table that lists the precedence and associativity of all the operators in C++. Operators are listed top to bottom, in descending precedence. Descending precedence refers to the priority of evaluation. Considering an expression, an operator which is listed on some row will be evaluated prior to any operator that is listed on a row further below it. Operators that are in the same cell (there may be several rows of operators listed in a cell) are evaluated with the same precedence, in the given direction. An operator's precedence is unaffected by overloading. The syntax of expressions in C++ is specified by a context-free grammar.

A precedence table, while mostly adequate, cannot resolve a few details. In particular, note that the ternary operator allows any arbitrary expression as its middle operand, despite being listed as having higher precedence than the assignment and comma operators. Thus  $a ? b, c : d$  is interpreted as  $a ? (b, c) : d$ , and not as the meaningless  $(a ? b), (c : d)$ . Also, note that the immediate, unparenthesized result of a C cast expression cannot be the operand of `sizeof`. Therefore, `sizeof (int) * x` is interpreted as `(sizeof(int)) * x` and not `sizeof ((int) *x)`.

| Precedence | Operator         | Description                            | Associativity |
|------------|------------------|----------------------------------------|---------------|
| 1          | ::               | Scope resolution (C++ only)            | Left-to-right |
| 2          | ++               | Suffix increment                       |               |
|            | --               | Suffix decrement                       |               |
|            | ()               | Function call                          |               |
|            | []               | Array subscripting                     |               |
|            | .                | Element selection by reference         |               |
|            | ->               | Element selection through pointer      |               |
| 3          | ++               | Prefix increment                       | Right-to-left |
|            | --               | Prefix decrement                       |               |
|            | +                | Unary plus                             |               |
|            | -                | Unary minus                            |               |
|            | !                | Logical NOT                            |               |
|            | ~                | Logical bitwise NOT                    |               |
|            | *                | Indirection (dereference)              |               |
|            | &                | Address-of                             |               |
|            | sizeof           | Size-of                                |               |
|            | new, new[]       | Dynamic memory allocation (C++ only)   |               |
|            | delete, delete[] | Dynamic memory deallocation (C++ only) |               |
| 4          | .*               | Pointer to member (C++ only)           | Left-to-right |
|            | ->*              | Pointer to member (C++ only)           |               |
| 5          | *                | multiplication                         | Left-to-right |
|            | /                | Division                               |               |
|            | %                | Modulus (remainder)                    |               |

|    |     |                                   |               |
|----|-----|-----------------------------------|---------------|
| 6  | +   | Addition                          |               |
|    | -   | Subtraction                       |               |
| 7  | <<  | Bitwise left shift                |               |
|    | >>  | Bitwise right shift               |               |
| 8  | <   | Less than                         |               |
|    | <=  | Less than or equal to             |               |
|    | >   | Greater than                      |               |
|    | >=  | Greater than or equal to          |               |
| 9  | ==  | Equal to                          |               |
|    | !=  | Not equal to                      |               |
| 10 | &   | Bitwise AND                       |               |
| 11 | ^   | Bitwise XOR (exclusive or)        |               |
| 12 |     | Bitwise OR (inclusive or)         |               |
| 13 | &&  | Logical AND                       |               |
| 14 |     | Logical OR                        |               |
| 15 | ?:  | Ternary conditional               |               |
| 16 | =   | Direct assignment                 | Right-to-left |
|    | +=  | Assignment by sum                 |               |
|    | -=  | Assignment by difference          |               |
|    | *=  | Assignment by product             |               |
|    | /=  | Assignment by quotient            |               |
|    | %=  | Assignment by remainder           |               |
|    | <<= | Assignment by bitwise left shift  |               |
|    | >>= | Assignment by bitwise right shift |               |
|    | &=  | Assignment by bitwise AND         |               |
|    | ^=  | Assignment by bitwise XOR         |               |
|    | =   | Assignment by bitwise OR          |               |
| 17 | ,   | Comma                             | Left-to-right |

### 3.6 MANIPULATORS

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display. These are the operators that can be used with insertion (<<) and extraction (>>) operators to manipulate or format the data in a desired way. There are certain manipulators that are used with << operator to display the output in a particular format whereas certain manipulator are used with >> operator to input the data in a desired form. The manipulators are used to set field widths, set precision, inserting new lines, skipping white spaces etc. In a single I/O statement, we can have more than one manipulator which can be as shown below:

```
cout<<manip1<<var1<<manip2<<var2;
```

```
cout<<manip1<<manip2<<var1;
```

### List of Manipulators

| Manipulator                     | Purpose                                 | Input/Output | Header file |
|---------------------------------|-----------------------------------------|--------------|-------------|
| setfill (int <i>ch</i> )        | set the field character to <i>ch</i>    | Output       | iomanip.h   |
| setiosflags(fmtflags <i>f</i> ) | turn on the flags specified in <i>f</i> | Input/Output | iomanip.h   |
| setprecision (int <i>p</i> )    | set the number of digits of precision   | Output       | iomanip.h   |
| setw (int <i>w</i> )            | set the field width to <i>w</i>         | Output       | iomanip.h   |
| showbase                        | turns on <b>showbase</b> flag           | Output       | iostream.h  |
| showpoint                       | turns on <b>showpoint</b> flag          | Output       | iostream.h  |
| showpos                         | turns on <b>showpos</b> flag            | Output       | iostream.h  |
| skipws                          | turns on <b>skipws</b> flag             | Input        | iostream.h  |
| unitbuf                         | turns on <b>unitbuf</b> flag            | Output       | fstream.h   |
| uppercase                       | turns on <b>uppercase</b> flag          | Output       | lstream.h   |
| ws                              | skip leading whitespace                 | Input        | iostream.h  |

**setw:** This manipulator sets the minimum field width on output. The syntax is:

```
setw(x)
```

#### Example:

```
#include <iostream.h>
using namespace std;
#include <iomanip.h>
void main()
{
 int x1=12345,x2= 23456, x3=7892;
 cout<<setw(8) << "Exforsys" <<setw(20) <<
 "Values" <<endl
```

```
<<setw(8) << "E1234567" <<setw(20)<< x1 <<endl
<<setw(8) << "S1234567" <<setw(20)<< x2 <<endl
<<setw(8) << "A1234567" <<setw(20)<< x3 <<endl;
}
```

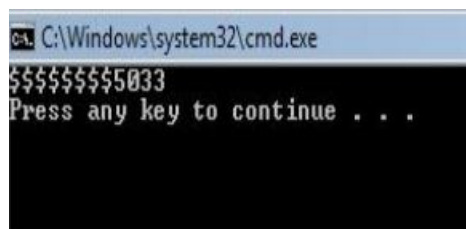
Here `setw` causes the number or string that follows it to be printed within a field of `x` characters wide and `x` is the argument set in `setw` manipulator.

**setfill:** This is used after the `setw` manipulator. If a value does not entirely fill a field, then the character specified in the `setfill` argument of the manipulator is used for filling the fields.

#### Example:

```
#include <iostream.h>
using namespace std;
#include <iomanip.h>
void main()
{
 cout<<setw(10) <<setfill('$') << 50 << 33 <<endl;
}
```

#### Output:



This is because the `setw` sets 10 for the width of the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with \$ symbol which is specified in the `setfill` argument.

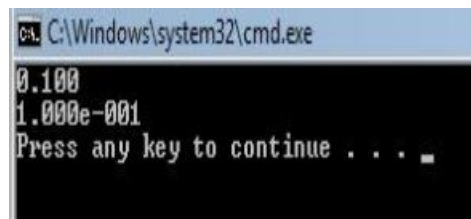
**setprecision:** The `setprecision` manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords `fixed` or `scientific` are appropriately used before the `setprecision` manipulator. The keyword `fixed` before the `setprecision` manipulator prints the floating point number in fixed notation. The keyword `scientific`, before the `setprecision` manipulator, prints the floating point number in scientific notation.

**Example:**

```
#include <iostream.h>
using namespace std;
#include <iomanip.h>
void main()
{
 float x = 0.1;
 cout<< fixed <<setprecision(3) << x <<endl;
 cout<< scientific << x <<endl;
}
```

**Output:**

The first `cout` statement contains `fixed` notation and the `setprecision` contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first `cout` statement as 0.100. The second `cout` produces the output in scientific notation. The default value is used since no `setprecision` value is provided.

**showbase:** When the `showbase` format flag is set, numerical values are prefixed with their C++ base format prefix when inserted into the stream. These prefixes are, 0x for hexadecimal values, 0 for octal values and no prefix for decimal-base values.

This option can be unset with the `noshowbase` manipulator, inserting all numerical values without base format prefixes.

The `showbase` flag is not set in standard streams on initialization.

**Example:**

```
#include <iostream.h>
using namespace std;
int main()
{
 int n;
 n=20;
 cout<< hex <<showbase<< n <<endl;
 cout<< hex <<noshowbase<< n <<endl;
 return 0;
}
```

**Output:** The execution of this example displays something similar to,



**showpoint:** When the showpoint format flag is set, the decimal point is always written for floating point values inserted into the stream, even for whole numbers. Following the decimal point, as many digits as necessary are written to match the precision internal setting for the stream (if any). This flag can be unset with the noshowpoint manipulator. When the showpoint format flag is not set, the decimal point is only written for non-whole numbers. The precision setting can be modified using the precision() member function of the stream.

The showpoint flag is not set in standard streams on initialization.

**Example:**

```
#include<iostream.h>
using namespace std;
int main()
{
 double a, b, pi;
 a=30.0;
 b=10000.0;
 pi=3.1416;
 cout.precision (5);
 cout<<showpoint<< a << '\t' << b << '\t' << pi
 <<endl;
 cout<<noshowpoint<< a << '\t' << b << '\t' << pi
 <<endl;
 return 0;
}
```

**Output:**

|        |        |        |
|--------|--------|--------|
| 30.000 | 10000. | 3.1416 |
| 30     | 10000  | 3.1416 |

**showpos:** When the showpos format flag is set, a plus sign (+) precedes every non-negative numerical value inserted into the stream, including zeros.

This flag can be unset with the noshowpos manipulator.

The showpos flag is not set in standard streams on initialization.

#### Example:

```
#include<iostream.h>
using namespace std;
int main()
{
 signedint p, z, n;
 p=1;
 z=0;
 n=-1;
 cout<<showpos<< p <<'\t'<< z <<'\t'<< n <<endl;
 cout<<noshowpos<< p <<'\t'<< z <<'\t'<< n <<endl;
 return 0;
}
```

#### Output:

|    |    |    |
|----|----|----|
| +1 | +0 | -1 |
| 1  | 0  | -1 |

**skipws:** When the skipws format flag is set, as many whitespace characters as necessary are read and discarded from the stream until a non-whitespace character is found before every extraction operation. Tab spaces, carriage returns and blank spaces are all considered whitespaces.

This flag can be unset with the noskipws manipulator, forcing extraction operations to consider leading whitespaces as part of the content to be extracted.

The skipws flag is set in standard streams on initialization.

#### Example:

```
#include <iostream.h>
#include <sstream.h>
using namespace std;
```



```
int main ()
{
char a, b, c;
istringstream iss(" 123");
iss>>skipws>> a >> b >> c;
cout<< a << b << c <<endl;
iss.seekg(0);
iss>>noskipws>> a >> b >> c;
cout<< a << b << c <<endl;
return 0;
}
```

**Output:**



```
123
1
```

**unitbuf:** When the unitbuf flag is set, the associated buffer is flushed after each insertion operation.

This flag can be unset with the no unitbuf manipulator, not forcing flushes after every insertion.

The unitbuf flag is not set in standard streams on initialization.

**Example:**

```
#include <fstream.h>
using namespace std;
int main()
{
ofstream outfile("test.txt");
outfile<<unitbuf<< "Test" << "file" <<endl;
outfile.close();
return 0;
}
```

**uppercase:** When the uppercase format flag is set, uppercase (capital) letters are used instead of lowercase for representations on insert operations involving stream-generated letters, like some hexadecimal representations and numerical base prefixes.

This flag can be unset with the nouppercase manipulator, not forcing the use of uppercase for generated letters.

The uppercase flag is not set in standard streams on initialization.

**Example:**

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

```
using namespace std;
int main()
{
 cout<<showbase<< hex;
 cout<< uppercase << 77 <<endl;
 cout<<nouppercase<< 77 <<endl;
 return 0;
}
```

Output:

```
0X4D
0x4d
```

**ws:**Extracts as many whitespace characters as possible from the current position in the input sequence. The extraction stops as soon as a non-whitespace character is found. These whitespace characters extracted are not stored in any variable.

**Example:**

```
#include <iostream.h>
#include <sstream.h>
using namespace std;
int main()
{
 char a[10], b[10];
 istringstream("one \n \t two");
 iss>>noskipws;
 iss>> a >>ws>> b;
 cout<< a << "," << b <<endl;
 return 0;
}
```



## CHECK YOUR PROGRESS

2.Fill in the blanks.

- The \_\_\_\_\_ rules of a language specify which \_\_\_\_\_ is evaluated first when two operators with different precedence are \_\_\_\_\_ in an expression.
- \_\_\_\_\_ are operators used in C++ for formatting output.

- c) The \_\_\_\_ manipulator sets the minimum field width on output.
- d) The \_\_\_\_ manipulator is used after the setw manipulator.
- e) The \_\_\_\_\_ manipulator is used with floating point numbers.
- f) When the \_\_\_\_\_ format flag is set, the decimal point is always written for \_\_\_\_\_ values inserted into the stream, even for whole numbers.
- g) When the \_\_\_\_\_ flag is set, the associated buffer is flushed after each \_\_\_\_\_ operation.
- h) The \_\_\_\_ extracts as many \_\_\_\_\_ characters as possible from the current position in the input sequence.

---

### 3.7 LET US SUM UP

---

- Using the C++ operators we can easily use and manipulate a C++ entity to get the desired output.
- C++ Operators are special symbols used for specific purposes.
- These operators perform arithmetic (numeric) operations.
- Type casting operators allow us to convert a datum of a given type to another.
- The comma operator gives left to right evaluation of expressions.
- Relational operators are used to test the relation between two values.
- The manipulators are used to set field widths, set precision, inserting new lines, skipping white spaces etc.
- In addition to standard assignment operator shown above, C++ also support compound assignment operators.

- The precedence rules of a language specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.



### 3.8 ANSWERS TO CHECK YOUR PROGRESS

---

1.
  - (a) Binary
  - (b) false, true
  - (c) logical
  - (d) unary
  - (e) assignment, right, left
  - (f) Bitwise
  - (g) Ternary
  - (h) Sizeof
  - (i) address, address
  - (j) indirection
2.
  - a) precedence, operator, adjacent
  - b) Manipulators
  - c) setw
  - d) setfill
  - e) setprecision
  - f) showpoint, floating point
  - g) unitbuf, insertion
  - h) ws, whitespace



### 3.9 FURTHER READINGS

---

Programming with C++, Second Edition

- John R. Hubbard

Tata McGraw-Hill Edition

C++ The Complete Reference

- Herbert Schildt

Tata McGraw-Hill Edition



### 3.10 MODEL QUESTIONS

---

1. What are C++ operators? What is the use of operators in C++?
2. What are arithmetic operators in C++? Name them and write the function of each
3. How many relational/comparison operators are there in C++ and what are they?
4. What are unary operators in C++?
5. What is the function of assignment operators?
6. Describe the function of the conditional operators in C++ with an example.
7. What do mean by precedence and Associativity rules of operators?
8. What are C++ manipulators? Name any five along with their purpose.
9. Mention the header files needed for the following code-

```
void main ()
{
cout<<setw (10) <<"name"<<setw (10) <<"marks"<<endl;
cout<<setw (10) <<"ram"<<setw (10) <<"95"<<endl;
cout<<setw (10) <<"hari"<<setw (10) <<"85"<<endl;
getch ();
}
```

## **UNIT- 4 DECISION AND CONTROL STRUCTURE**

### **UNIT STRUCTURE**

- 4.1 Learning Objectives
- 4.2 Introduction
- 4.3 Conditional statements
  - 4.3.1 if statement
  - 4.3.2 if-else statement
  - 4.3.3 switch-case statement
- 4.4 Loop statements
  - 4.4.1 for loop
  - 4.4.2 while loop
  - 4.4.3 do-while loop
- 4.5 Breaking control statements
  - 4.5.1 break statement
  - 4.5.2 continue statement
  - 4.5.3 goto statement
- 4.6 Let Us Sum Up
- 4.7 Answers to check your progress
- 4.8 Further Readings
- 4.9 Model Questions

---

### **4.1 LEARNING OBJECTIVES**

---

After going through this unit, you will be able to:

- Understand the basic concept of branching
- Use if, if-else and switch statements in a program
- Understand the importance of loop & control constructs in a program
- Use for, while and do...while statements in a program
- Define break, continue and goto statements

---

## 4.2 INTRODUCTION

---

In this chapter you will learn about the different decision structures like if, if-else, switch-case and also about the control structures like for loop, while loop and do while loop. Finally the control breaking statements like break, continue and goto are explained later in this unit.

---

## 4.3 CONDITIONAL STATEMENTS

---

Till now we have used sequence control structure in our programs in which the various steps are executed sequentially to get the desired output. But this may not happen always. Sometimes we might need to take a decision for executing an instruction. In simple terms the program might need to execute a set of instructions for one condition and a set of another instruction for another condition. Let us consider some examples. If it is raining I would remain indoors. If I don't get a ticket in train, I would go by bus. If I get the money, I would buy a laptop. All the decisions in the examples given above depends on some condition

To implement this kind of decision control instructions we have to use if or if-else statements in our program.

---

### 4.3.1 if STATEMENT

---

The if statement checks a certain condition. If the condition is true then the block of statements following the if is executed; otherwise it executes the optional statements. The braces { and } are used to group declaration and statements into a compound statement or a block. The basic simple structure of the if statement is shown below:

```
if(expression)
{
 statement;
 statement;
 statement;
}
```

The expression is any valid expression which is examined to a numeric value.

For example: if(n==1)

```
{
```

```
 cout<<"Have a great day ahead";
 }
```

If the given condition is true then the computer will print the message "Have a great day ahead" otherwise it will skip this statement.

Note that that there is no semicolon after the if expression.

The expression

```
 if(x>y);
 {
 temp=x;
 }
```

is wrong because the compiler will interpret it as

```
 if(x>y)
 ;
 temp=x which is meaningless.
```

**Program1:** Program to illustrate the use of if statement

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num;
 clrscr();
 cout<<"Enter a number less than 10:";
 cin>>num;
 if(num<10)
 {
 cout<<"Good Morning !";
 }
 getch();
}
```

---

### 4.3.2 If-else STATEMENT

---

The basic structure of if-else statement is shown below:



```
if(expression)
{
 statement1;
}
else
{
 statement2;
}
```

In if-else statement either of the two statements are executed. If the given expression is true then statement1 is executed else statement2 is executed.

Some of the sample if-else structures are shown below

1) if(expression)

```
{

}
else
{

}
```

2) if(expression){

```
 if(expression){

 }
 else{

 }
}
```

else{

```


 }

3) if(expression){
 if(expression){

 }
 else{

 }
 }
 else{
 if(expression){

 }
 }
 else{

 }
}
```

**Program2:** Program to find whether a given number is even or not.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num;
 clrscr();
 cout<<"Enter a number: \n";
 if(num%2==0)
```

```
 {
 cout<<"The number is even";
 }
 else
 {
 cout<<"The number is odd";
 }
 getch();
}
```

Output of the above program

Enter a number

56

The number is even

**Program3:** Program to find the largest value among any four numbers

```
#include<iostream.h>
#include<conio.h>
void main()
{
 float a,b,c,d;
 cout<<"Enter any four numbers\n";
 cin>>a>>b>>c>>d;
 if(a>b){
 if(a>c){
 if(a>d)
 cout<<"Largest = "<<a;
 else
 cout<<"Largest= "<<d;
 }
 }
 else
 {
 if(c>d)
```

```
 cout<<"Largest = "<<c;
 else
 cout<<"Largest= "<<d;
 }
}
else
{
 if(b>c){
 if(b>d)
 cout<<"Largest= "<<b;
 else
 cout<<"Largest= "<<d;
 }
 else{
 if(c>d)
 cout<<"Largest = "<<c;
 else
 cout<<"Largest= "<<d;
 }
 }
getch();
}
```

Output of the above program

Enter any four numbers

56

32

78

12

Largest=78

---

### 4.3.3. switch STATEMENT

The switch statement allows us to make a decision from a multiple choice of decisions.

The syntax of the switch statement is :

```
switch(expression)
{
case1:
{
 statement1;
}
case2:
{
 statement2;
}

case_n:
{
 statement;
}
default:
 statement;
}
```

The expression, whose value is being compared, may be any valid expression including the value of the variable, an arithmetic expression, a logical comparison, a bitwise expression or the return value of a function call but not a floating point type. The keyword case is followed by an integer or a character constants but it cannot be an expression. It can neither be a floating point number or a character string. The default case is optional and should be used according to the program's specific requirement.

When we run a program containing switch first the expression following the switch keyword is checked. The value given by the expression is matched one by one against all the values followed by the case statements. When a match is found, the program executes the case statement following it as well as also the subsequent cases and the default case are also executed. If no match is found then only the statements following the default statement is executed.

**Program4:** Program to illustrate the use of switch statement

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int i;
 clrscr();
 cout<<"Enter number 1, 2 or 3 \n";
 cin>>i;
 switch(i)
 {
 case 1: cout<<"I am in case 1\n";
 case 2: cout<<"I am in case 2\n";
 case 3: cout<<"I am in case 3\n";
 default:cout<<"I am in default\n";
 }
 getch();
}
```

Output of the above program

```
Enter number 1, 2 or 3
2
I am in case 2
I am in case 3
I am in default
```

This is not what we expected. As the value matches with case 2 it should execute only the statement "I am in case 2". In order to overcome this problem the break statement is used. The break statement causes an immediate exit from the switch construct. It is used at the end of each case statement however it is not necessary to use a break statement after a default statement. The break statement is discussed later in this unit.

Thus the general construct of a switch statement is:

```
switch(expression)
```

```
{
 case1:
 {
 statement1;
 }
 break;
 case2:
 {
 statement2;
 }
 break;

 case_n:
 {
 statement;
 }
 break;
 default:
 statement;
}
```

Now Program4 would be like this:

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int i;
 clrscr();
 cout<<"Enter number 1, 2 or 3 \n";
 cin>>i;
 switch(i)
```

```
{
 case 1: cout<<"I am in case 1\n";
 case 2: cout<<"I am in case 2\n";
 case 3: cout<<"I am in case 3\n";
 default: cout<<"I am in default\n";
}
getch();
}
```

Output of the above program

Enter number 1, 2 or 3

2

I am in case 2

**Program5:** Program to generate a simple calculator

```
#include<iostream.h>
#include<conio.h>
void main()
{
 float a,b;
 char opr;
 clrscr();
 cout<<"Enter any two numbers:";
 cin>>a>>b;
 cout<<"Enter the operation to be performed(+,-,*,/) \n";
 cin>>opr;
 switch(opr)
 {
 case '+': cout<<a<<'+<<b<< '='<<(a+b);
 break;
 case '-': cout<<a<<'-<<b<< '='<<(a-b);
 break;
 case '*': cout<<a<< '*<<b<< '='<<(a*b);
```



```
 break;
 case '/': if(b==0)
 cout<<"Operation not possible";
 else
 cout<<a<<'/'<<b<<'='<<(a/b);
 break;
 default:cout<<"Wrong choice";
}
getch();
}
```

Output of the above program

Enter any two numbers:

45

54

Enter the operation to be performed(+, -, \*, /)

+

45+54=99

**Program6:** Program to check whether a character is vowel or not.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 char ch;
 clrscr();
 cout<<"Enter a character \n";
 cin>>ch;
 switch(ch)
 {
 case 'a':
 case 'A':
```

```
 case 'e':
 case 'E':
 case 'i':
 case 'I':
 case 'o':
 case 'O':
 case 'u':
 case 'U': cout<<"It is a vowel";
 break;
 default:cout<<"It is not a vowel";
}
getch();
}
```

Output of the above program

Enter a character

E

It is a vowel



### CHECK YOUR PROGRESS

- 1) State whether the following expressions are true or false:
  - a) The switch statement allows us to make a decision from a multiple choice of decisions.
  - b) if-else is a loop statement
  - c) Conditional expressions are mainly used for decision-making
  - d) if, if-else and while are the three conditional statements
  - e) The statement in the default case of switch statement is executed when all other cases doesn't match with a certain condition.

## 4.4 LOOP STATEMENTS

The loop statements are used to execute a certain set of instructions repeatedly either a fixed number of times or until a given condition is being satisfied. There are three ways in which a particular part of a program can be repeated. They are:

- a) Using a for statement
  - b) Using a while statement
  - c) Using a do-while statement
- 

#### 4.4.1 for LOOP

---

The for loop consist of three expressions. The first expression is to initialize the loop counter, the second expression is used to check the number of times it should be repeated and the third expression is to increment the loop counter.

The syntax of the for loop is:

```
for(initial value; test; increment)
{
 statement1;
 statement2;

}
```

**Program7:** Program to print the numbers & their sum which are divisible by 5 within a certain limit

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int l,i,sum=0;
 clrscr();
 cout<<"Enter the limit:";
 cin>>l;
 cout<<"\n The numbers divisible by 5 are:\n";
 for(i=1;i<=l;i++)
 {
 if(i%5==0)
 {
 cout<<i<<"\n";
 sum=sum+i;
 }
 }
}
```

```

 }
}
cout<<"\n The sum is: "<<sum;
getch();
}

```

Output of the above program:

Enter the limit: 20

The numbers divisible by 5 are:

5

10

15

20

The sum is: 50

**Program8:** Program to print the following pattern:

```

1
1 2
1 2 3
1 2 3 4

```

```

#include<iostream.h>
#include<conio.h>
void main()
{
 int line_no,i,j;
 clrscr();
 cout<<"\n Enter the desired number of lines<=10:";
 cin>>line_no;
 cout<<"Desired pattern is:\n\n";
 for(i=1;i<=line_no;i++)
 {
 for(j=1;j<=i;j++)
 cout<<j<<"\t";
 cout<<"\n";
 }
 getch();
}

```

```
}
```

Output of the above program

Enter the desired number of lines<=10:

6

Desired pattern is:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

---

#### 4.4.2 while STATEMENT

In programming sometimes we might want to print a message for a fixed number of times or we might want to calculate the gross salary of ten employees. In such cases while loop is the ideal solution. The general form of the while loop is:

```
while(expression)
{
 statement1;
 statement2;

}
```

The expression is any valid C++ expression including the value of a variable, a unary or binary expression or the value returned by a function. The expression actually tests a certain condition.

**Program9:** Program to find the factorial of a number

```
#include<iostream.h>
#include<conio.h>
void main()
```

```
{
unsigned long int n,
fact=1;
clrscr();
cout<<"Enter the number:\n";
cin>>n;
while(n>0)
{
 fact=fact*n;
 n--;
}
cout<<"Factorial of the number is:\n"<<fact;
getch();
}
```

Output of the above program

Enter the number:

10

Factorial of the number is:

3628800

**Program10:** Program to find the sum of digits of a number

```
#include<iostream.h>
#include<conio.h>
void main()
{
int n,sum=0,r;
clrscr();
cout<<"Enter the number:\n";
cin>>n;
while(n>0)
{
 r=n%10;
```

```
 sum=sum+r;
 n=n/10;

 }
 cout<<"The sum of digit is: \n"<<sum;
 getch();
}
```

Output of the above program

Enter the number:

536

The sum of digit is:

14

---

#### 4.4.3 do-while STATEMENT

Another loop used in C++ is the do-while statement. When a programmer is sure about a certain test condition the do-while loop is used as it enters the loop for atleast one time. This is where there lies a difference between while and do-while loop because while loop enters the loop only when it satisfies a certain condition. The general syntax of the do-while loop is:

```
do{
 statement1;
 statement2;

}while(expression);
```

The expression is any valid C++ expression.

**Program11:**Program to find the sum of even numbers

Sum = 2 + 4 + 6 +.....+ n

```
#include<iostream.h>
#include<conio.h>
```

```
void main()
{
 int num,sum=0,l;
 clrscr();
 cout<<"Enter the limit\n";
 cin>>l;
 num=2;
 do
 {
 sum=sum+num;
 num=num+2;
 }
 while(num<=l);
 cout<<"The sum is :\n"<<sum;
 getch();
}
```

Output of the above program

Enter the limit

20

The sum is:

110

**Program12:** Program to find the odd numbers which are less than 50

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num=1,l;
 clrscr();
 cout<<"Enter the limit:\n";
 cin>>l;
 cout<<"The odd numbers are:\n";
```



```
do
{
 if(num%2!=0)
 {
 cout<<"\n"<<num;
 }
 num++;
}while(num<=l);
getch();
}
```

Output of the above program

Enter the limit:

20

The odd numbers are:

1

3

5

7

9

11

13

15

17

19

**CHECK YOUR PROGRESS**

2) What will be the output of the following program?

```
main()
{
 int j;
 while(j<=10)
 {
 cout<<j;
 j=j+1;
 }
}
```

3) What will be the output of the following program?

```
main()
{
 for(int i=0;i<10;i++);
 {
 cout<<"Hello";
 }
 getch();
}
```

4) Choose the correct option:

i) A do-while loop is useful when we want that the statements within the loop must be executed:

- a) only once
- b) atleast once
- c) more than once
- d) none of the above

ii) Which of the following loop statements consist of three expressions?

- a) while loop
- b) do-while loop
- c) for loop
- d) None of the above

**4.5 BREAKING CONTROL STATEMENTS**

C++ allows the use of three types of control break statements:

- 1) break statement
- 2) continue statement
- 3) goto statement

**4.5.1 break STATEMENT**

Normally to terminate control from the loop structure of switch case statements, break statements are used. break statement must be used after every case statement of a switch structure otherwise the control will transfer to the subsequent case conditions.

The general format of break statement is:

```
break;
```

**a) break statement used with a switch-case structure**

```
#include<iostream.h>
#include<conio.h>
void main()
{
 char ch;
 clrscr();
 cout<<"Enter a character: \n";
 cin>>ch;
 switch(ch)
 {
 case 'r':
 case 'R': cout<<"The colour is red";
 break;
 case 'b':
 case 'B': cout<<"The colour is blue";
 break;
 case 'g':
 case 'G': cout<<"The colour is green";
 break;
 default: cout<<"Wrong choice";
 }
 getch();
}
```

Output of the above program

Enter a character:

G

The colour is green

## 2) break statement used in a while loop

A break statement can also be used in other loops.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num,i=1,l;
 clrscr();
 cout<<"Enter the limit";
 cin>>l;
 while(i<=l)
 {
 cout<<"Enter a number:";
 cin>>num;
 if(num<=0)
 {
 cout<<"Zero or negative value found \n";
 break;
 }
 i++;
 }
 getch();
}
```

Output of the above program

Enter the limit

7

Enter a number: 1

Enter a number: 2

Enter a number: 3

Enter a number: 4

Enter a number: -7

Zero or negative value found

This program processes only positive integers. As soon as a zero or a negative value is entered the control exits from the loop displaying the message "Zero or negative value found".

---

### 4.5.2 continue STATEMENT

---

The continue statement is used to continue the same operations even if an error is encountered. The general syntax of the continue statement is:

```
continue;
```

The continue statement is just the inverse of the break statement.

Use of continue statement in a while loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num,i=1,l;
 clrscr();
 cout<<"Enter the limit";
 cin>>l;
 while(i<=l)
 {
 cout<<"Enter a number:";
 cin>>num;
 if(num<=0)
 {
 cout<<"Zero or negative value found \n";
 continue;
 }
 i++;
 }
}
```

```
getch();
}
```

Output of the above program

Enter the limit

7

Enter a number: 1

Enter a number: 2

Enter a number: 3

Enter a number: 4

Enter a number: -7

Zero or negative value found

Enter a number: 5

Enter a number: 6

The above program processes only positive integers. But here when the program encounters a negative value, it displays the message “Zero or negative value found” and continues the same loop as long as the given condition is satisfied.

---

### 4.5.3 goto STATEMENT

---

The goto statement is used to alter the execution of a program sequence by transferring the control of a program to some different part of the program. The general syntax of goto statement is:

```
goto label;
```

The label is a C++ identifier that is used to label the destination where the program control is to be transferred.

The two ways of using goto statement is:

**a) UNCONDITIONAL goto-** In unconditional goto the control of a program is transferred to any other part of the program without checking any condition

```
//unconditional goto statement
#include<iostream.h>
void main()
{
```

```
start :
cout<<"I am doing a C++ program";
goto start;
}
```

A good programmer may never use an unconditional goto statement as it might create some problems. In the above program as there is no condition, the program will become a never ending process.

**b) CONDITIONAL goto-** In conditional goto statement the control is transferred to some other part of a program after satisfying certain conditions.

```
//conditional goto statement
#include<iostream.h>
#include<conio.h>
void main()
{
int n;
clrscr();
cout<<"Enter a number:\n";
cin>>n;
if(n%2==0)
 goto output1;
else
 goto output2;
output1:
 cout<<"The number is even";
 goto stop;
output2:
 cout<<"The number is odd";
 stop:
getch();
}
```

//use of goto statement inside the while loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int num, i=1,l;
 clrscr();
 cout<<"Enter the limit";
 cin>>l;
 while(i<=l)
 {
 cout<<"Enter a number:";
 cin>>num;
 if(num<=0)
 {
 cout<<"Zero or negative value found \n";
 goto error;
 }
 i++;
 }
 error:
 cout<<"Input data error \n";
 getch();
}
```

Output of the above program

Enter the limit

7

Enter a number: 0

Zero or negative value found

Input data error





## CHECK YOUR PROGRESS

5) Choose the correct option

i) The break statement is used to exit from:

- a) an if statement
- b) a for loop
- c) a program
- d) the main() function

ii) The \_\_\_\_\_ statement is used to alter the program execution sequence by transferring control to some other part of the program.

- a) continue statement
- b) break statement
- c) goto statement
- d) none of the above

iii) Which statement is used to repeat the same operations once again even if it checks the error?

- a) continue statement
- b) goto statement
- c) break statement
- d) none of the above

---

## 4.6 LET US SUM UP

- **Conditional Statements** are used in programs which need to take a certain decision based on some condition.
  - There are three conditional statements- **if**, **if-else** and **switch case** statement
  - **if statement** checks a certain condition and executes the following statement if it is true.
  - In **if-else statement** either of the two statements is executed.
  - **switch statement** allows to make a decision from multiple choice of decisions.
  - **Loop statements** are used to execute a certain set of instructions repeatedly until a given condition is satisfied.
  - There are three loop statements- **for**, **while** and **do-while**.
-

- The **for** loop consists of three expressions- initializing the index value, checking the condition whether to continue or discontinue the loop and lastly to increment the index value for further iteration.
- **while** loop is used when there is no certainty whether the loop will execute or not.
- **do-while** loop enters the loop for at least once and then checks whether the given condition is true or not.
- **break** statement is used to terminate and exit from a particular point of a program.
- **continue** statement is used to continue the same operations even if an error is encountered.
- The **goto** statement is used to alter the sequence of a program by transferring the control to other part of the program.



## 4.7 ANSWERS TO CHECK YOUR PROGRESS

---

- 1)a) True      b)False      c)True d)False      e)True  
2) No output  
3) "Hello" will be executed only once  
4) i)b) atleast once      ii)c) for loop  
5)i) c) a program      ii)c) goto statement      iii)a)continue



## 4.8 FURTHER READINGS

---

Programming with C++, Second Edition

- John R. Hubbard

Tata McGraw-Hill Edition

C++ The Complete Reference

- Herbert Schildt

Tata McGraw-Hill Edition



## 4.9 MODEL QUESTIONS

---

- 1) What are conditional statements? What are the different types of conditional statements?
- 2) What is the difference between if and if-else statement?
- 3) Explain the switch case statement with example.
- 4) What are looping statements? What are the different types of loop statements?
- 5) How the while loop differ from the do-while loop?
- 6) When do we use a switch case statement in a program?
- 7) How do for loop differ from while loop?
- 8) Explain the use of break, continue and goto statement in a program?
- 9) Write down the general syntax for declaring:
  - a) for loop      b) while loop      c) do-while loop
- 10) Write a program that prints the numbers and its cube from 1 to 10 using the following control statements:
  - a) if-else      b) for loop      c) while loop      d) do-while loop
- 11) Write a program to check whether a number is Armstrong number or not.
- 12) Write a program to find the multiplication table of any number within a certain limit.
- 13) Write a program to reverse a number.
- 14) Write a program to find the sum of the following series:
$$\text{Sum} = 1 + 3 + 5 + 7 + \dots + n$$
- 15) Write a program to generate the Fibonacci series.
- 16) Write a program to check whether a year is leap year or not.
- 17) Write a program to find the roots of a quadratic equation.
- 18) Write a program to generate the following pattern:
$$\begin{array}{r} 1=1 \\ 1+2=3 \\ 1+2+3=6 \end{array}$$
- 19) Write a program to check whether a number is prime or not.
- 20) Write a program to calculate the occurrences of positive numbers, negative numbers and zeros in a stream of data terminated by some specific value.

## **UNIT- 5 Arrays, Pointers, Structure and Union**

### **UNIT STRUCTURE**

- 5.1 Learning Objectives
- 5.2 Introduction
- 5.3 Arrays
  - 5.3.1 Array Declaration
  - 5.3.2 Array Initialization
  - 5.3.3 Processing with Array
  - 5.3.4 Arrays and Functions
  - 5.3.5 Multidimensional Arrays
  - 5.3.6 Character Array
  - 5.3.7 Initializing Character Array
- 5.4 Pointers
  - 5.4.1 Pointer Operators
  - 5.4.2 Pointers & Arrays
- 5.5 Structures
  - 5.5.1 Declaration of a Structure
  - 5.5.2 Initialization of a Structure
  - 5.5.3 Accessing Structure Variables
  - 5.5.4 Structure within a Structure (Nested Structure)
  - 5.5.5 Arrays of Structure
- 5.6 Unions
- 5.7 Let Us Sum Up
- 5.8 Answers to check your progress
- 5.9 Further Readings
- 5.10 Model Questions

---

### **5.1 LEARNING OBJECTIVES**

---

After going through this unit, you will be able to:

- Understand the basics of arrays
- Understand multidimensional arrays

- Understand character arrays
- Write programs on arrays
- Understand pointers
- Understand relationship between pointers and arrays
- Understand structures
- Understand nested structures
- Understand union data type

---

## **5.2 INTRODUCTION**

---

In this unit you will learn arrays, declaration of arrays, initialization of arrays and also multidimensional arrays. Later there is introduction to pointers and structure as well as their relationship with arrays. Lastly the union data type is discussed.

---

## **5.3 ARRAYS**

---

An array is a collection of homogenous data objects which are stored in contiguous memory locations under a common variable name. The individual data objects are called elements of an array. Array can also be defined as a collection of similar data types which have a single name followed by an index. A subscript or an index is a positive integer value that determines the position of an element in an array. Depending upon the number of subscript used, arrays can be either one dimensional or multidimensional.

---

### **5.3.1 SINGLE DIMENSIONAL ARRAY**

---

The array which requires only one subscript to access the elements of an array are the single or one dimensional array. A single dimensional array can be declared as:

`data_type array_name[size]`

Here `data_type` refers to the type of data (integer, float, character), `array_name` refers to the name of the array and `size` refers to the number of elements in the array. The size of an array is also called dimension of an array.

For example: `int value[10];`

Here value is an array of type integer and size 10.

Some more examples of declaring a one dimensional array:

```
int a[10];
```

```
char stu_name[20];
```

```
float x[100]
```

In the above examples, the first one is an integer array 'a' of size 10; the second one is a character stu\_name of size 20 and the third one is a floating point array of size 100.

---

### **5.3.2 INITIALIZATION OF SINGLE DIMENSIONAL ARRAY**

---

The general format of single dimensional array initialization is:

```
data_type array_name[size] = {element1, element2,.....element_n};
```

Here data\_type refers to the type of data, array\_name refers to the name of the array and size refers to the size of the array. Finally the elements of the array are placed one after the other within the braces ended with a semicolon.

Example:

```
int x[10]= {1, 2,3,4,5,6,7,8,9,10};
```

```
char name[5]={'V','i','d','y','a'};
```

```
float value[5]={2.6,0,-8.7,4.5,9.6};
```

The results of the above array elements are:

```
x[0]=1
```

```
x[1]=2
```

```
x[2]=3
```

```
x[3]=4
```

```
x[4]=5
```

```
x[5]=6
```

```
x[6]=7
```

```
x[7]=8
```

```
x[8]=9
```

```
x[9]=10
```

```
name[0]='V'
```

```
name[1]='i'
name[2]='d'
name[3]='y'
name[4]='a'
```

```
values[0]=2.6
values[1]=0
values[2]=-8.7
values[3]=4.5
values[4]=9.6
```

In arrays, the first element is always placed in the 0<sup>th</sup> place i.e, the array index always starts from 0. So if there are n elements in an array the array index will start from 0 to n-1.

---

### 5.3.3 ACCESSING SINGLE DIMENSIONAL ARRAY ELEMENTS

---

The syntax for accessing the elements of a single dimensional array is:

```
array_name[subscript];
```

For example the elements of the array can be referred to as: a[0], a[1], a[2] and so on.

The elements of a single dimensional array are stored in contiguous blocks of memory i.e the elements are always stored next to each other.

The memory representation of a single dimensional array is:

```
int a[10];
```

|      |      |      |      |      |       |      |      |      |      |
|------|------|------|------|------|-------|------|------|------|------|
| a[0] | a[1] | a[2] | a[3] | a[4] | a [5] | a[6] | a[7] | a[8] | a[9] |
| 10   | 20   | 30   | 40   | 50   | 60    | 70   | 80   | 90   | 100  |

Examples of programs using single dimensional arrays

**Program1:** Program to display the elements of a given array

```
#include<iostream.h>
#include<conio.h>
void main()
{
```

```
int p[10]={10,20,30,40,50,60,70,80,90,100};
int i;
clrscr();
cout<<"The array is:\n";
for(i=0;i<10;i++)
{
 cout<<p[i]<<"\t";
}
getch();
}
```

Output of the above program is:

The array is:

10      20      30      40      50      60      70      80      90      100

**Program2:** Program to read n numbers from the keyboard and display the elements of the array.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int p[10],i,n;
 clrscr();
 cout<<"Enter the number of elements in the array:\n";
 cin>>n;
 cout<<"Enter the elements of the array:\n";
 for(i=0;i<n;i++)
 {
 cin>>p[i];
 }
 cout<<"The elements of the array are:\n";
 for(i=0;i<n;i++)
 {
```



```
 cout<<p[i]<<'\t';
 }
 getch();
}
```

Output of the above program

Enter the number of elements in the array:

5

Enter the elements of the array:

2

4

6

8

10

The elements of the array are:

2      4      6      8      10

**Program3:** Program to read n numbers from the keyboard and find the minimum element of the array

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int a[10],i,n,min;
 clrscr();
 cout<<"Enter the number of elements in the array:\n";
 cin>>n;
 cout<<"Enter the elements of the array:\n";
 for(i=0;i<n;i++)
 {
 cin>>a[i];
 }
 cout<<"The elements of the array are:\n";
 for(i=0;i<n;i++)
```

```
{
 cout<<a[i]<<'\\t';
}
min=a[0];
for(i=1;i<n;i++)
{
 if(a[i]<min)
 {
 min=a[i];
 }
}
cout<<"\\n\\nThe minimum element is:"<<min;
getch();
}
```

Output of the above program

Enter the number of elements in the array:

5

Enter the elements of the array:

54

46

32

99

76

The elements of the array are:

54    46    32    99    76

The minimum element is: 32

**Program4:** Program to read n numbers from the keyboard and display the elements of the array in sorted order.

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

```
#include<conio.h>
```

```
void main()
```

```
{
int a[10],i,j,n,temp;
clrscr();
cout<<"Enter the number of elements in the array:\n";
cin>>n;
cout<<"Enter the elements of the array:\n";
for(i=0;i<n;i++)
{
 cin>>a[i];
}
cout<<"The elements of the array are:\n";
for(i=0;i<n;i++)
{
 cout<<a[i]<<"\t";
}
for(i=0;i<n;i++)
{
 for(j=i+1;j<n;j++)
 {
 if(a[i]>a[j])
 {
 temp=a[i];
 a[i]=a[j];
 a[j]=temp;
 }
 }
}
cout<<"\nThe elements of the array in sorted order:\n";
for(i=0;i<n;i++)
{
 cout<<a[i]<<"\t";
}
getch();
```

```
}
```

Output of the above program

Enter the number of elements in the array:

10

Enter the elements of the array:

45

12

76

22

90

11

55

68

44

88

The elements of the array are:

45    12    76    22    90    11    55    68    44    88

The elements of the array in sorted order:

11    12    22    44    45    55    68    76    88    90

---

### 5.3.4 MULTIDIMENSIONAL ARRAYS

An array of arrays is said to be as multidimensional array. In multidimensional array there may be n subscripts/index. In general arrays with more than three dimensions are not used. Here we shall discuss arrays with two dimensions. Sometimes some data fit better in a table with several rows and columns. This can be constructed by using two-dimensional arrays.

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form:

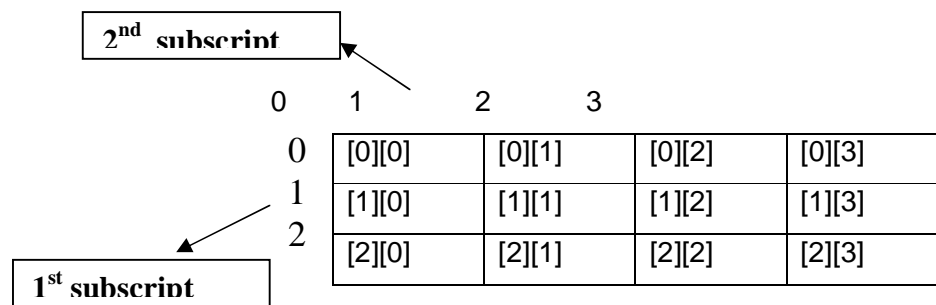
data\_type    array\_name[row\_size][column\_size];

For example: int x[3][4];

```
float matrix[20][25];
```

The first line declares x as an integer array with 3 rows and 4 columns and the second line declares a matrix as a floating-point array with 20 rows and 25 columns.

Graphically, `int x[3][4]` can be depicted as follows:



You can see that for `[3][4]` 2D array size; the total array size (the total array elements) is equal to 12.

Hence: For n rows and m columns, the total size equal to mn

### 5.3.5 Initialization of two dimensional arrays

Just like the one-dimensional array, a two dimensional array can also be initialized. For example, the previous first array declaration can be rewritten along with initial assignments in any of the following ways:

```
int x[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Or

```
int x[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

The results of the initial assignments in both cases are as follows:

```
x[0][0]=1 x[0][1]=2 x[0][2]=3 x[0][3]=4
x[1][0]=5 x[1][1]=6 x[1][2]=7 x[1][3]=8
x[2][0]=9 x[2][1]=10 x[2][2]=11 x[2][3]=12
```

Let us see some examples of programs using two dimensional arrays.

**Program5:** Program to initialize a set of numbers in a two dimensional array and to display the content of the array.

```
#include<iostream.h>
#include<conio.h>

void main()
{
 int i,j;

 int a[3][4]={{1,2,3,4},
 {5,6,7,8},
 {9,10,11,12}};

 clrscr();

 cout<<"Contents of the array are:\n";
 for(i=0;i<3;i++) //row size is 3
 {
 for(j=0;j<4;j++) //column size is 4
 {
 cout<<a[i][j]<<"\t";

 }
 cout<<"\n";
 }

 getch();
}
```

Output of the above program:

Contents of the array are:

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

9 10 11 12

**Program6:** Program to display the contents of a two dimensional array whose size is 3x3.

```
#include<iostream.h>

#include<conio.h>

void main()

{

int i,j;

int a[3][3];

clrscr();

cout<<"Enter the elements of the array:";

for(i=0;i<3;i++)

{

 for(j=0;j<3;j++)

 {

 cin>>a[i][j];

 }

}

cout<<"Contents of the array are:\n";

for(i=0;i<3;i++)

{

 for(j=0;j<3;j++)

 {

 cout<<a[i][j]<<"t";

 }

 cout<<"\n";
```

```
}
getch();
}
```

Output of the above program:

Enter the elements of the array:

10  
20  
30  
40  
50  
60  
70  
80  
90

Contents of the array are:

|    |    |    |
|----|----|----|
| 10 | 20 | 30 |
| 40 | 50 | 60 |
| 70 | 80 | 90 |

**Program7:** Program to find the sum of two 2x2 matrices.

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int i,j;
```

```
int a[2][2],b[2][2],c[2][2];
```

```
clrscr();
```



```
cout<<"Enter the elements of the first matrix:\n";
for(i=0;i<2;i++)
{
 for(j=0;j<2;j++)
 {
 cin>>a[i][j];
 }
}
cout<<"Enter the elements of the second matrix:\n";
for(i=0;i<2;i++)
{
 for(j=0;j<2;j++)
 {
 cin>>b[i][j];
 }
}
cout<<"Contents of first matrix:\n";
for(i=0;i<2;i++)
{
 for(j=0;j<2;j++)
 {
 cout<<a[i][j]<<"\t";
 }
 cout<<"\n";
}
cout<<"Contents of second matrix:\n";
```

```
for(i=0;i<2;i++)
{
 for(j=0;j<2;j++)
 {
 cout<<b[i][j]<<"t";

 }
 cout<<"\n";
}
cout<<"The sum of two matrices is:\n";
for(i=0;i<2;i++)
{
 for(j=0;j<2;j++)
 {
 c[i][j]=a[i][j]+b[i][j];
 cout<<c[i][j]<<"t";

 }
 cout<<"\n";
}
getch();
}
```

Output of the above program:

Enter the elements of the first matrix:

1

2

3

4

Enter the elements of the second matrix:

5

6

7

8

Contents of first matrix:

1 2

3 4

Contents of second matrix:

5 6

7 8

The sum of two matrices is:

6 8

10 12

---

### **5.3.6 CHARACTER ARRAYS (STRINGS)**

---

An array of characters is known as a character array. A character array can be declared as:

```
char name[20];
```

Here name is a character array whose size is 20.

An array of characters in which the last character is terminated by a null character represents a string. The null character is denoted by '\0'.

### **5.3.7 INITIALIZING CHARACTER ARRAYS**

---

Like an integer or a floating point array, the character array can also be initialized. For example,

```
char name[7]="Sameer";
```

The elements would be assigned to each of the character array position in the following way:

```
name[0]='S';
```

```
name[1]='a';
```

```
name[2]='m';
```

```
name[3]='e';
```

```
name[4]='e';
```

```
name[5]='r';
```

```
name[6]='\0';
```

The null character will be added automatically by the C++ compiler provided there is enough space to accommodate the character.

The basic structure of character array is:

| name[0] | name[1] | name[2] | name[3] | name[4] | name[5] | name[6] |
|---------|---------|---------|---------|---------|---------|---------|
| S       | a       | m       | e       | e       | r       | \0      |

There are a number of predefined string functions for manipulating strings in different manner. These functions are defined in the string.h file. Some of the string functions are:

strcat- Appends a copy of a string to another string

strcmp- Compares two strings

strlen- Counts the number of characters in a string

**Program8:** Program to read a string and display it.

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
char str[20];
```

```
clrscr();
```

```
cout<<"Enter a string :\n";
```

```
cin>>str;

cout<<"The string is: ";

cout<<str;

getch();

}
```

Output of the above program

Enter a string:

Welcome

The string is: Welcome

**Program9:** Program to read a string & display the string and the length of the string using strlen() function

```
#include<iostream.h>

#include<conio.h>

#include<string.h>

void main()

{

char str[20];

int l;

clrscr();

cout<<"Enter a string :\n";

cin>>str;

cout<<"The string is:";

cout<<str;

l=strlen(str);

cout<<"\nThe length of the string is: "<<l;

getch();
```

```
}
```

Output of the above program

Enter a string:

Program

The string is: Program

The length of the string is: 7

**Program10:** Program to read two strings & display the concatenation of the two strings.

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

```
#include<conio.h>
```

```
#include<string.h>
```

```
void main()
```

```
{
```

```
char str1[20],str2[20];
```

```
clrscr();
```

```
cout<<"Enter the first string :\n";
```

```
cin>>str1;
```

```
cout<<"Enter the second string :\n";
```

```
cin>>str2;
```

```
cout<<"\nConcatenated string is: "<<strcat(str1,str2);
```

```
getch();
```

```
}
```

Output of the above program:

Enter the first string:

class

Enter the second string:

room

Concatenated string is: classroom



### CHECK YOUR PROGRESS

- 1) Arrays are sets of values of the same type which have a single name followed by an \_\_\_\_\_.
- 2) A subscript or an index is a positive integer value that determines the \_\_\_\_\_ of an element in an array.
- 3) The array index always starts from \_\_\_\_\_ to n-1, where n is the maximum size of the array declared by the programmer.
- 4) A two dimensional array will require \_\_\_\_\_ pairs of square brackets.
- 5) A character string is always terminated by \_\_\_\_\_ character.
- 6) An array of characters is called a \_\_\_\_\_.
- 7) Which of the following is not a valid array declaration?
  - i) `int value[20];`
  - ii) `float y[30];`
  - iii) `char [s];`
  - iv) `char name[10];`
- 8) Strings are always specified in
  - i) double quotes
  - ii) braces
  - iii) single quotes
  - iv) square brackets

---

## 5.4 POINTERS

A pointer is a variable that is used to store a memory address. The address is the location of the variable in the memory. Pointers help in allocating memory dynamically. Pointers improve execution time and saves space.

The general syntax of declaring a pointer variable is:

`data_type *pointer_variable`

where `data_type` is the type of the pointer variable such as integer, character and floating point number variable etc., and `pointer_variable` is any valid C++ identifier.

For example:     `int *value;`  
                  `char *name;`

When a pointer variable is declared, the variable name must be preceded by an asterisk (\*). This identifies the variable as a pointer.

---

### 5.4.1 POINTER OPERATORS

---

A pointer variable consists of two parts i) the pointer operator and ii) the address operator. A pointer operator can be represented by a combination of \* (asterisk) with a variable. The \* operator is also called indirection operator. It returns the contents of the memory location pointed to.

An address operator can be represented by a combination of & (ampersand) with a pointer variable. The & operator is a unary operator. The unary operator returns the address of the memory where a variable is located.

To understand the concept of these two operators let us consider the following statements-

```
int *x;
int c=100;

int p;

x=&c;

p=*x;
```

Here in the first statement x is a pointer variable of type integer. The second statement is a simple assignment statement where 100 is assigned to c and then in the next statement another variable p of type integer is declared.

Now,

```
x=&c;
```

Here & operator returns the memory address of the variable c to x.

Lastly,

```
p=*x;
```

Here \* operator returns the content of the pointer x and variable p will contain value 100 as the pointer x contain value 100 at its memory location. Let us consider a program which illustrates the working of pointers.

**Program11:** Program to display the contents of a pointer.

```
#include<iostream.h>
#include<conio.h>
void main()
{
```



```
int x,y;
int*ptr;
x=10;
clrscr();
ptr=&x;
y=*ptr;
cout<<"Let us \n";
cout<<"Contents of x:"<<x;
cout<<"\nContents of pointer variable: "<<ptr;
cout<<"\nContents of y: "<<y;
getch();
}
```

Output of the above program

Contents of x: 10

Contents of the pointer variable: 0x8fd6fff4

Contents of y: 10

**Program12:** Program to display the address and the contents of a pointer variable.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int x;
int *ptr;
x=20;
clrscr();
ptr=&x;
cout<<"Let us \n";
cout<<"Contents of x:"<<x;
cout<<"\nContents of pointer variable: "<<*ptr;
cout<<"\nAddress of pointer variable: "<<ptr;
getch();
}
```

```
}
```

Output of the above program

Contents of x: 20

Contents of the pointer variable: 20

Address of the pointer variable: 0x8fd6fff4

---

## 5.4.2 POINTERS AND ARRAYS

---

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations,

```
int x[20];
```

```
int *p;
```

The following assignment operation would be valid:

```
p=x;
```

After that, p and x would be equivalent and would have the same properties. The only difference is that we could change the value of pointer p by another one, whereas x will always point to the first of the 20 elements of type int with which it was defined. Therefore, unlike p, which is an ordinary pointer, x is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

```
x=p;
```

Because x is an array, so it operates as a constant pointer, and we cannot assign values to constants.

The following is a valid assignment:

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

The address of the zeroth element of x is assigned to the pointer variable p.

If the pointer is incremented to the next data element, then the address of the incremented value of the pointer will be same as the value of the next element.

```
p++==value[1];
```

Array subscripting is defined in terms of pointer arithmetic. That is the expression

```
x[i]
```

is defined to be the same as

\*((x) + (i))

which is to say the same as

\* (&(x) [0] + (i))

**Program12:** Program to display the contents of an array using pointer arithmetic.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int value[5],n;
 int *p;
 clrscr();
 p=value;
 *p=10;
 p++;
 *p=20;
 p=&value[2];
 *p=30;
 p=value+3;
 *p=40;
 p=value;
 *(p+4)=50;
 for(n=0;n<5;n++)
 {
 cout<<value[n]<<" ";
 }
 getch();
}
```

Output of the above program

10,20,30,40,50



## CHECK YOUR PROGRESS

9) Fill in the blanks

- i) Pointers help in allocating \_\_\_\_\_ dynamically.
- ii) An address operator can be represented by a combination of \_\_\_\_\_ with a pointer variable.
- iii) The \_\_\_\_\_ operator returns the address of the memory where a variable is located.
- iv) The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the \_\_\_\_\_ of the first element that it points to,

---

## 5.5 STRUCTURES

---

Structure is a collection of heterogeneous data types. The individual components of a structure which are called fields or members can be accessed and processed separately. Array & structures have two main differences. Firstly in arrays, all the elements are of same data type whereas in structure all the elements may be of different data types. Secondly each component of an array is referred to by its position whereas each component in structure has a unique name

The general syntax of declaring a structure is:

```
struct user_defined_name{
 data_type member 1;
 data_type member 2;

 data_type member n;
};
```

A structure definition is specified by the keyword `struct`. The keyword `struct` is followed by a user defined name of the structure surrounded by braces, which describes the members of the structure. The braces are terminated by a semi colon.

For example to declare a structure containing details of an employee is:

```
struct employee
{
 int code;
 char name[20];
 int age;
}
```

Similarly to declare a structure containing details of a book is:

```
struct book
{
 int book_code;
 char name[20];
 float price;
}
```

---

### **5.5.1 DECLARING STRUCTURE VARIABLES**

---

To use structure in a program, a structure variable needs to be declared. The general syntax of declaring a structure variable is:

```
struct structure_variable;
```

For example to declare a structure variable of a structure 'date' can be declared as :

```
struct date d1;
```

Here d1 is a variable of type structure whose name is 'date'.

---

### **5.5.2 INITIALIZATION OF STRUCTURE**

---

A structure can be initialized in the same way as any other data type in C++. To illustrate let us consider an example of a structure containing details of a student such as roll number, name, age.

The structure will be initialized as:

```
struct student s1={101,"Sneha",20};
```

Here s1 is a structure variable.

---

### 5.5.3 ACCESSING STRUCTURE VARIABLES

---

The structure variables can be accessed by using the period or dot operator ('.').

For example date is a structure consisting of three members which can be referred in program as:

```
struct date
{
 int day;
 int month;
 int year;
}

void main()
{
 struct date d1;
 d1.day;
 d1.month;
 d1.year;
}
```

The members of the structure can be assigned values as:

```
today.day=24;
today.month=3;
today.year=2012;
```

Reading a structure will be as follows:

```
cin>>today.day;
cin>>today.month;
cin>>today.year;
```

Writing a structure will be as follows:

```
cout<<today.day;
cout<<today.month;
cout<<today.year;
```

Let us consider some programs to illustrate the use of structure.

**Program 13:** Program to assign some values to the members of a structure and to display the structure.

```
#include<iostream.h>
#include<conio.h>
struct student
{
int roll_no;
char name[20];
int age;
};
void main()
{
clrscr();
struct student s1={162,"Pranab",21};
cout<<endl<<"Roll Number: "<<s1.roll_no;
cout<<endl<<"Name: "<<s1.name;
cout<<endl<<"Age: "<<s1.age;
getch();
}
```

Output of the above program:

Roll Number: 162

Name: Pranab

Age: 21

**Program14:** Program to display the details of an employee using structures.

```
#include<iostream.h>
#include<conio.h>
struct employee
{
 int code;
 char name[20];
 int age;
 float salary;
};
```

```
void main()
{
 struct employee emp;
 clrscr();
 cout<<"Enter the details of the employee- "<<endl;
 cout<<"Code: ";
 cin>>emp.code;
 cout<<endl<<"Name: ";
 cin>>emp.name;
 cout<<endl<<"Age: ";
 cin>>emp.age;
 cout<<endl<<"Salary: ";
 cin>>emp.salary;
 cout<<"The details of the employee:"<<endl;
 cout<<"Code: "<<emp.code<<endl;
 cout<<"Name: "<<emp.name<<endl;
 cout<<"Age: "<<emp.age<<endl;
 cout<<"Salary: "<<emp.salary<<endl;
 getch();
}
```

Output of the above program:

Enter the details of the employee-

Code: 1090

Name: Shekhar

Age: 27

Salary: 20000

The details of the employee:

Code: 1090

Name: Shekhar

Age: 27

Salary: 20000



---

### 5.5.4 STRUCTURE WITHIN A STRUCTURE (Nested Structure)

---

It is possible to use a structure as a member of another structure or in other words we can say that there can be a structure within a structure. Such type of structure is called nested structure. The nested structure can be declared as :

```
struct first
{
 int a;
 float b;
 char s;
};
struct second
{
 int p;
 struct first one;
}
```

**Program15:** Program to calculate the total course fee of a student who has enrolled in two courses using nested structure.

```
#include<iostream.h>
#include<conio.h>
struct course
{
 int course_no;
 int course_fee;
};
struct student
{
 int stud_rollno;
 struct course c1;
 struct course c2;
};
void main()
```

```
{
 int x;
 student s1;
 clrscr();
 s1.stud_rollno=234;
 s1.c1.course_no=111;
 s1.c1.course_fee=5000;
 s1.c2.course_no=114;
 s1.c2.course_fee=6000;
 x=s1.c1.course_fee + s1.c2.course_fee;
 cout<<"\nStudent Roll Number: "<<s1.stud_rollno<<"\nTotal Course Fee: "<<x;
 getch();
}
```

Output of the above program:

Student Roll Number: 234

Total Course Fee: 11000

---

### 5.5.5 ARRAY OF STRUCTURES

---

Just as arrays of basic types such as integers and floats are allowed in C, so are arrays of structures. An array of structures is simply an array in which each element is a structure of the same type. The referencing and subscripting of these arrays (also called structure arrays) follow the same rules as simple arrays. An array of structures is declared as:

```
struct course
{
 char course_name[20];
 int course_code;
 float course_fee;
};
course c[20];
```

The c[20] is a structure variable. It may accommodate the structure of a course up to 20. Each record may be accessed & processed separately like individual elements of an array. Let us see some examples.

**Program16:** Program to read n employees information & display them.

```
#include<iostream.h>
#include<conio.h>
struct employee
{
 char name[20];
 int code;
 int age;
 int dept_no;
 float salary;
};
void main()
{
 struct employee emp[10];
 int i,n;
 clrscr();
 cout<<"Enter the number of employees: ";
 cin>>n;
 for(i=0;i<n;i++)
 {
 cout<<endl<<"Name :";
 cin>>emp[i].name;
 cout<<endl<<"Code : ";
 cin>>emp[i].code;
 cout<<endl<<"Age: ";
 cin>>emp[i].age;
 cout<<endl<<"Department Number: ";
 cin>>emp[i].dept_no;
 cout<<endl<<"Salary: ";
```

```
 cin>>emp[i].salary;
 }
 cout<<endl<<"Details of the employees:";
 for(i=0;i<n;i++)
 {
 cout<<endl<<"Name : "<<emp[i].name;
 cout<<endl<<"Code : "<<emp[i].code;
 cout<<endl<<"Age: " <<emp[i].age;
 cout<<endl<<"Department Number: " <<emp[i].dept_no;
 cout<<endl<<"Salary: " <<emp[i].salary;
 }
 getch();
}
```

Output of the above program:

Enter the number of employees: 4

Name : Kaushik

Code : 236

Age: 25

Department Number: 1106

Salary: 15000

Name : Mohit

Code : 654

Age: 27

Department Number: 1104

Salary: 13000

Name : Neha

Code : 765

Age: 27

Department Number: 1103

Salary: 18000

Name : Shruti

Code : 342

Age: 25

Department Number: 1102

Salary: 10000

Details of the employees:

Name : Kaushik

Code : 236

Age: 25

Department Number: 1106

Salary: 15000

Name : Mohit

Code : 654

Age: 27

Department Number: 1104

Salary: 13000

Name : Neha

Code : 765

Age: 27

Department Number: 1103

Salary: 18000

Name : Shruti

Code : 342

Age: 25

Department Number: 1102

Salary: 10000

---

## 5.6 UNIONS

---

Unions are similar to structure but they differ from each other by the way how data is stored and retrieved. Unions are declared in the same fashion as structures, but have a fundamental difference. Only one item within the union can be used at any time, because the memory allocated for each item inside the union is in a shared memory location.

The syntax of a union declaration is:

```
union user_defined_name
{
 data_type member1;
 data_type member2;
```

```


data_type member_n;
}
```

The keyword union is essential. data\_type is any valid C++ data type such as int, float and char.

A structure can be a member of a union and also a union can be a member of structure

Example:

```
union class
{
 int one;
 float two;
 char value;
};
union u;
```

where u is a union variable.

**Program 17:** Program to initialize the members of a union and display the contents of the union.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 union sample
 {
 int x;
 float y;
 };
 union sample u;
 clrscr();
 u.x=45;
 u.y=18.54;
 cout<<"x= "<<u.x<<endl;
```

```
 cout<<"y= "<<u.y<<endl;
 getch();
}
```

Output of the above program:

x=20972

y=18.54

In the above program the members of union are int and float. The float values are stored correctly as well as displayed correctly. But the integer values are not displayed correctly because union holds only a value for one data type which requires a larger storage among their members.



## CHECK YOUR PROGRESS

10) State true or false

- a) Structures are a collection of homogenous elements.
- b) Each member of a structure is specified by a variable name with a period and the member name.
- c) An array is used to store dissimilar elements and a structure to store similar elements.

11) When a structure is declared as the member of another structure, it is called \_\_\_\_\_ structure.

12) The \_\_\_\_\_ data type stores values of different types in a single location.

---

## 5.7 LET US SUM UP

- An **array** is a collection of homogenous data objects which are stored in contiguous memory locations under a common variable name.
- The array which requires only one subscript to access the elements of an array are the **single** or **one dimensional array**.
- An array of arrays is said to be as **multidimensional array**.
- In **multidimensional array** there may be **n** subscripts/index.
- A **two dimensional array** has two subscripts/indexes. The first subscript refers to the row, and the second, to the column.
- An array of characters is known as a **character array**.
- An array of characters in which the last character is terminated by a null character represents a **string**.
- A **pointer** is a variable that is used to store a memory address.
- A pointer variable consists of two parts - the **pointer operator** and the **address operator**.
- **Arrays** and **pointers** are of the same concept.
- In fact, the **identifier** of an array is equivalent to the **address** of its first element, as a pointer is equivalent to the **address** of the **first element** that it points to.



- **Structure** is a collection of heterogeneous data types.
- There can be a structure within a structure, called **nested structure**.
- An **array of structures** is simply an array in which each element is a structure of the same type.
- **Unions** are similar to structure but they differ from each other by the way how data is stored and retrieved.
- A **structure** can be a member of a union and also a **union** can be a member of structure.



## 5.8 ANSWERS TO CHECK YOUR PROGRESS

---

- 1) index
- 2) position
- 3) 0
- 4) two
- 5) null
- 6) string
- 7) iii) char [s]
- 8) i) double quotes
- 9) i) memory ii) ampersand (&) iii) ampersand(&) iv) address
- 10) a) False b) True c) False
- 11) nested
- 12) union



## **5.10 MODEL QUESTIONS**

---

- 1) What is an array? How an array is declared and initialized?
- 2) Explain multidimensional array with the help of an example.
- 3) What is a character array?
- 4) What is a pointer?
- 5) Explain the use of pointer in a program.
- 6) With the help of an example explain how will you relate a pointer and an array?
- 7) What is a structure? How a structure is declared and initialized?
- 8) Differentiate between array and structures.
- 9) Explain nested structure with the help of an example.
- 10) What is a union data type? How does union differ from structure?
- 11) Write a program to reverse the elements of an array.
- 12) Write a program to find the maximum element of an array.
- 13) Write a program to find the sum of elements of a 3x3 matrix.
- 14) Write a program to check whether a given string is palindrome or not.
- 15) Write a program to read the following information for n employees and display the record of the employee whose salary is maximum:
  - Employee name
  - Employee code
  - Designation
  - Salary
  - Years of experience
  - Age
- 16) Write a program to read the following information for n patients and display the record of the desired patient whose name is supplied by the user:
  - Name of the patient

Code of the Patient

Sex

Age

Ward number

Bed number

Nature of illness

Date of admission

- 17) Write a program to find the transpose of a matrix.
- 18) Write a program to find the subtraction of two matrices.
- 19) Write a program to insert an element in an array at a specific position.
- 20) Write a program to delete an element in an array from a specific position.

## UNIT - 6: FUNCTIONS

### UNIT STRUCTURE

- 6.1 Learning Objectives
- 6.2 Introduction
- 6.3 main() Function
- 6.4 Components of a Function
  - 6.4.1 Function Prototype
  - 6.4.2 Function Definition
  - 6.4.3 Function Parameters
  - 6.4.4 Function Call
- 6.5 Passing Arguments
- 6.6 Type of Functions
- 6.7 Inline Functions
- 6.8 Function Overloading
- 6.9 Let Us Sum Up
- 6.10 Answers To Check Your Progress
- 6.11 Further Readings
- 6.12 Model Questions

---

### 6.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to:

- know what are C++ functions
- learn about the different components of a function
- make function calls
- learn how to pass arguments to a function
- know about the different types of functions
- know what function overloading is

---

### 6.2 INTRODUCTION

---

In the previous few units, we got to know quite a few important aspects of the C++ programming language. Most useful programs are much larger than the programs that we have considered so far.

To make large programs manageable, programmers modularize them into sub-programs. These sub-programs are called functions. They can be compiled and tested separately and reused in different programs. This modularization is characteristic of successful Object-Oriented software.

We are already familiar with the concept of functions in the C language which are almost same in all programming languages excepting some minor differences in the syntax. A repeated group of instructions in a program can be organized as a function. Instead of writing the same lines of code, functions can be invoked wherever they are required. A function is a sequence of instructions written in a particular programming language like C, C++ etc. that perform some specific tasks as specified by the user.

---

### 6.3 main() FUNCTION

---

We have already learnt that any C program starts with the main() function. Let us first discuss the use of the main( ) function of the C++ language. Every C++ program requires a function named main(). In fact, we can think of the complete program itself as being made up of the main() function together with all the other functions that are called either directly or indirectly from it. The program starts by calling main(). Since main() is a function with return type int, it is normal to end its block with

```
return 0;
```

although most compilers do not require this. Some compilers allow it to be omitted but will issue a warning when it is. The value of the integer that is returned to the operating system should be the number of errors counted; the value 0 is the default.

The basic structure of the main() function in C++ is as follows:

```
void main()
{
 // the statements of the C++ program
}
```

But since the `main()` function in C++ returns an integer to the operating system, the syntax of the `main()` function in C++ is as follows:

```
int main()
{
.....
.....
return 0; // may return 0 or 1
}
```

---

## 6.4 COMPONENTS OF A FUNCTION

---

A C++ function is composed of different parts and each part plays a unique role in carrying out the operations of a function successfully. These separate parts of a C++ function operate individually to carry out the overall task of the function.

The different parts of a C++ function are as follows-

1. Function prototype
2. Function definition
3. Function parameters
4. Function call

---

### 6.4.1 FUNCTION PROTOTYPE

---

One of the most important features of C++ is the function prototypes. A function prototype tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls. In C++ all functions must be declared using a function prototype before they are used. Prototypes enable C++ to provide stronger type checking. When we use prototypes, the compiler can find and report any illegal type conversions between the type of arguments used to call a function and the type definition of its parameters. The compiler will also catch differences between the number of arguments used to call a function and the number of parameters in the function.

The general form of a function prototype is

```
return_type func_name(type param_name);
int main()
{
.....
}
```

The use of parameter name is optional. However, they enable the compiler to identify any type mismatches by name when an error occurs, so it is a good idea to include them.

---

## 6.4.2 FUNCTION DEFINITION

---

A function definition provides the actual body of the function.

### Defining a Function:

The general form of a C++ function definition is as follows:

```
return_type function_name(parameter list)
{
 // body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return type:** A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.
- **Function name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, we pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function body:** The function body contains a collection of statements that define what the function does.

### Example:



The following example shows how a function `max()` takes two parameters `num1` and `num2` and returns the maximum between them:

```
// function returning the max between two numbers
int max(int num1, int num2)
{
 // local variable declaration
 int result;
 if (num1 > num2)
 result = num1;
 else
 result = num2;
 return result;
}
```

In the above code, the `max` function holds the value of `num1` and `num2` as its arguments. Inside the function, an integer variable `result` is declared, and a test is made to find the greater number among `num1` and `num2`. The number with a greater value gets stored in the variable `result` and is returned to the calling statement that was used to call the `max` function.

---

### 6.4.3 FUNCTION PARAMETERS

---

A parameter is a special kind of variable, used in a function to refer to one of the pieces of data provided as input to the function. These pieces of data are called parameters. An ordered list of parameters is usually included in the definition of a function, so that, each time the function is called, its arguments for that call can be assigned to the corresponding parameters.

The parameters in a C++ function are used to exchange data between the calling and the called functions. These parameters that are used in any C++ program using functions can be divided into two types.

**Actual parameters**– These parameters are specified in the function call.

**Formal parameters**– These parameters are specified in the function declaration.

A parameter cannot be both a formal and an actual parameter, but both formal parameters and actual parameters can be either value parameters or variable parameters.

**Example-**

```
#include<stdio.h>
int main(void);
int add(int, int, int);
int main()
{
 int total;
 int num1 = 25;
 int num2 = 32;
 int num3 = 27;
 total = add(num1, num2, num3);
 printf("Total sum=", bill);
 exit (0);
}
int add(int num1, int num2, int num3)
{
 int total;
 total = num1 + num2 + num3;
 return total;
}
```

In the function main in the example above, num1, num2, and num3 are all actual parameters when used to call add. On the other hand, the corresponding variables in add (namely num1, num2 and num3, respectively) are all formal parameters because they appear in a function definition.

Although formal parameters are always variables (which does not mean that they are always variable parameters), actual parameters do not have to be variables. We can use numbers, expressions, or even function calls as actual parameters. Here are some examples of valid actual parameters in the function call to add:

```
total = add(25, 32, 27);
total = add(50+60, 25*2, 100-75);
```

---

**6.4.4 FUNCTION CALL**

---

We use C++ functions to carry out some specific tasks of that function. But in order to do so the call to that function has to be made in order for the function to execute properly providing us the desired results. A C++ function gets executed only when it is invoked or called.

The following example demonstrates how a C++ function call is made.

```
int sum (int, int); // function prototype
void main()
{
 int s;
 ;
 s=sum(3,6); // function call

}
```

In the above code, the statement `s=sum(3,6);` is a function call statement. When the compiler encounters a function call, the control is transferred to the function definition. Then the function is executed line by line as defined in the function definition and when the return statement is encountered, the resulting value is returned to the calling function. `sum(3,6)` function is called and value returned by this function is assigned to the variables.



## CHECK YOUR PROGRESS

1. Fill in the blanks.

- The compiler uses function \_\_\_\_\_ to validate function calls.
- \_\_\_\_\_ enable C++ to provide stronger type checking.
- A C++ function gets executed only when it is \_\_\_\_\_.
- A function \_\_\_\_\_ provides the actual body of the function.
- The return type is the \_\_\_\_\_ type of the value the function returns.

---

## 6.5 PASSING ARGUMENTS

---

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

As with local variables, we may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, we may use them as we do any other local variable.

There are two methods by which we can pass values to the function. These are:

- **Call by value**
- **Call by reference**
  
- **Call by value –**

This method copies the value of an argument into the formal parameter of the sub-routine. In this case, changes made to the parameter have no effect on the argument. By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Let us consider the following program.

```
#include<iostream.h>
#include<conio.h>

void increase(int);
void main()
{
 int num = 10;
 increase (num);
 cout<<num;
 getch();
}
void increase(int no)
{
```

```
 no = 15;
 }
```

The output will be: 10

In the above program we tried to change the value of 'num' after executing the program. 'num' is passed by value to the function definition so only a copy of the value of 'num' is copied 'no'. Any change in the function definition does not affect or change the original value of 'num'.

Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

- **Call by reference—**

This method is the second way of passing arguments to a function. In this method, the address of an argument is copied into the parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the arguments. We can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value. Of course we need to declare the parameters as pointer types.

```
#include<iostream.h>
#include<conio.h>

void increase(int&);
void main()
{
 int num = 10;
 cout<<"Old value="<<num;
 increase(num);
 cout<<"New value="<<num;
 getch();
}
void increase(int &no)
{
 no = no + 5;
}
```

The output will be:

Old value= 10

New value= 15

When the function is called, 'no' will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument.

---

## 6.6 TYPE OF FUNCTIONS

---

Depending on the arguments and return types of the functions they are classified into four categories. These are:

1. Function with no arguments and no return values.
2. Function with no arguments but with return values
3. Function with arguments and no return values.
4. Function with arguments and return values.

The type of function that we choose from among these four categories depends entirely upon the purpose of the function and the way it has to function and output the values.

### 1. **Function with no arguments and no return values –**

This type of function has no arguments i.e., it does not receive any data from the calling function. Also the function does not return any value. i.e., the calling function does not receive any data from the called function. In other words there is no data transfer between the calling and the called function. For example, in the following program we have used a function with no argument and no return value.

```
#include<iostream.h>
#include<conio.h>
voidshow();
void main()
{
 show();
 getch();
}
voidshow()
```

```
{
 cout<<"\n Hello World ";
}
```

**Output :** Hello World

As the function show() does not return any value, so the return type is void.

## 2. Function with no arguments but with return values –

In this type, the function returns a value to the calling function but it takes no argument. The following program illustrates this category:

```
#include<iostream.h>
#include<conio.h>
int adder(); //function declaration
void main()
{
 int s;
 clrscr();
 s=adder();
 printf("\nThe summation is: %d",s);
 getch();
}
int adder()
{
 int a,b,s;
 a=10;
 b=10;
 s=a+b;
 return s;
}
```

**Output:** The summation is: 20

As the function adder() is returning a value, so 20 is returned to the calling function and it is stored in the variable s and this value is displayed by the main() function.

## 3. Function with arguments and no return values –

In this category, the function receives data from the calling function through arguments, but does not return any value. The following program illustrates this category.

```
#include<iostream.h>
#include<conio.h>
void check(int); //function declaration
void main()
{
 int num;
 cout<<"\n Enter a number ";
 cin>>num;
 check(num);
 getch();
}
void check(int no) //function definition
{
 if(no>0)
 cout<<"\n Positive number";
 else
 cout<<"\n Not a Positive number";
}
```

In the above program, the number input is passed as argument to the function. The number is checked in the function definition part and displayed by the function `check()`.

#### 4. Function with arguments and return values –

In this category, the function accepts data from the calling function through arguments and after performing the required operations returns the resulting value back to the calling function.

In the example below, the function `check_positive()` accepts a number from the user and returns 1 if the number is positive otherwise it returns 0. The returned value is checked in the `main()` function, and an appropriate message is displayed.

```
#include<iostream.h>
#include<conio.h>
int check_positive(int); //function declaration
void main()
{
 int num, status;
 cout<<"\n Enter a number ";
 cin>>num;
 status = check_positive(num);
 if(status>0)
 cout<<"\nPositive number";
 else
 cout<<"\nNot a Positive number";
 getch();
}
```



```
 }
 int check_positive(int no)
 {
 if(no>0)
 return 1;
 else
 return 0;
 }
```

The value returned by `check_positive` is stored in the variable `status`. The value will be either 1 or 0 depending upon the number that is input.

---

## 6.7 INLINE FUNCTIONS

---

A function call involves substantial overhead. Extra time and space have to be used to invoke the function, pass parameters to it, allocate storage for its local variables, store the current variables and the location of execution in the main program, etc. In some cases, it is better to avoid all this by specifying the function to be inline. This tells the compiler to replace each call to the function with explicit code for the function. To the programmer, an inline function appears the same as an ordinary function, except for the use of the inline specifier.

The following program demonstrates the use of inline function:

```
#include<iostream.h>
#include<conio.h>
inline float area(float x)
{
 return x*x;
}
void main()
{
 float a;
 clrscr();
 cout<<"Enter length of a side of square (in
cm):";
 cin>>a;
 cout<<"\n Area of the square
:"<<area(a)<<"cm2";
 getch();
}
```

In the `main()` function, the statement

```
cout<<"\n Area of the square :"<<area(a)<<"cm2";
```

invokes the inline function `inline float area(float x)`. The body of the function is replaced at the point of its call. In this mechanism the execution time of the function `area(float x)` is less than the time required to establish a linkage between the caller (calling function) and callee (called function).

Even if the calling function is very large, the compiler copies the content of the inline function in the called function which reduces the program execution speed. So, in such a case inline function should not be used.

---

## 6.8 FUNCTION OVERLOADING

---

One way that C++ achieves polymorphism is through the use of function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation the functions that share the same name are said to be overloaded, and the process is referred to as function overloading. In general, to overload a function, we simply declare different versions of it. The compiler takes care of the rest. We need to observe one important restriction when overloading a function, which is that the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the types or number of their parameters. (return types do not provide sufficient information in all cases for the compiler to decide which function to use.) Of course, overloaded functions may differ in their returns types too.

The following program illustrates function overloading.

```
#include<iostream.h>
#include<conio.h>
#define pi 3.142
float area(float x); // function prototype
float area(float len, float bth);
float area(double radius);
void main()
```

```
{
 floata,m,n,p;
 clrscr();
 cout<<"Enter length of a side of square :";
 cin>>a;
 cout<<"\n Area of the square :"<<area(a);
 cout<<"\n Enter length and breadth of
rectangle :";
 cin>>m>>n;
 cout<<"\n Area of rectangle :"<<area(m,n);
 cout<<"\n Enter radius of circle :";
 cin>>p;
 cout<<"\n Area of circle :"<<area(p);
 getch();
}
float area(float x)
{
 return x*x;
}
float area(float len, float bth)
{
 return(len * bth);
}
float area(double radius)
{
 return(pi*radius*radius);
}
```

**RUN :**

```
Enter length of a side of square: 1.25
Area of the square: 1.5625
Enter length and breadth of rectangle: 5.3 2.6
Area of rectangle: 13.78
Enter radius of circle: 5.256
Area of circle : 27.625536
```

**In the above program, we have used three different functions:**

```
float area(float x);
float area(float len, float bth);
float area(double radius);
```

performing the same task, but with different data types. Here the function `area()` is overloaded. It is used for finding the area of square, rectangle and circle.



## CHECK YOUR PROGRESS

2. Fill in the blanks:

- a) The \_\_\_\_\_ in a C++ function are used to exchange data between the calling and the called functions.
- b) \_\_\_\_\_ parameters are specified in the function call.
- c) \_\_\_\_\_ parameters are specified in the function declaration.
- d) By default, C++ uses \_\_\_\_\_ to pass arguments.
- e) In \_\_\_\_\_ changes made to the parameter affect the arguments.
- f) Depending on the \_\_\_\_\_ and return types of the functions they are classified into \_\_\_\_\_ categories.
- g) An \_\_\_\_\_ function appears the same as an ordinary function, except for the use of the \_\_\_\_\_ specifier.
- h) In C++, two or more \_\_\_\_\_ can share the same name as long as their \_\_\_\_\_ declarations are different.

---

## 6.9 LET US SUM UP

---

- Every C++ program requires a function named main().
- A C++ function is composed of different parts and each part plays a unique role in carrying out the operations of a function successfully.
- A function prototype tells the compiler the name of the function, the data type returned by the function, the number of parameters, the types of the parameters, and the order in which these parameters are expected.

- A C++ function definition consists of a function header and a function body.
- Parameters are optional; that is, a function may contain no parameters.
- A parameter is a special kind of variable, used in a function to refer to one of the pieces of data provided as input to the function.
- Call by value copies the value of an argument into the formal parameter of the sub-routine.
- In Call by reference, the address of an argument is copied into the parameter.
- An inline function replaces each call to the function with explicit code for the function.
- C++ achieves polymorphism is through the use of function overloading.



## 6.10 ANSWERS TO CHECK YOUR PROGRESS

---

1.
  - (a) Prototypes
  - (b) Prototypes
  - (c) Invoked
  - (d) Definition
  - (e) Data
2.
  - (a) parameters
  - (b) Actual
  - (c) Formal
  - (d) Call by value
  - (e) call by reference
  - (f) arguments, four
  - (g) inline, inline
  - (h) functions, parameter



## 6.11 FURTHER READINGS

---

Programming with C++, Second Edition

- John R. Hubbard

Tata McGraw-Hill Edition

C++ The Complete Reference

- Herbert Schildt

Tata McGraw-Hill Edition

---



## 6.12 MODEL QUESTIONS

---

1. Describe the main() function of C++.
2. What is a C++ function?
3. How do we make function calls in C++?
4. Differentiate between actual and formal parameters in C++ with an example.
5. Explain giving an example the difference between call by value and call by reference.
6. How many types of C++ function are there?
7. What are inline functions? What is their use?
8. Explain with an example the way a function can be overloaded in C++.
9. Write a program to check the trigonometry  $\cos 2x = 2\cos^2 x - 1$ .

10. Write and test the following `power()` function that returns `x` raised to the power `n`, where `n` can be any integer:

```
double power (double x, int p);
```

11. Write and test the following `min` function that returns the smallest of four given integers. `int min(int, int, int, int);`

12. Write a C++ program to output the following Pascal's Triangle.

```

 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
 1 5 10 10 5 1
 1 6 15 20 15 6 1
 1 7 21 35 35 21 7 1
 1 8 28 56 70 56 28 8 1

```

## UNIT-7 : INTRODUCTION TO CLASSES AND OBJECT

### UNIT STRUCTURE

- 7.1 Learning Objective
- 7.2 Introduction
- 7.3 Classes in C++
- 7.4 Class Declaration
  - 7.4.1 Access Control in Class
- 7.5 Declaring Objects
  - 7.5.1 Accessing Class Members
- 7.6 Defining Member Functions
  - 7.6.1 Member Function inside a Class
  - 7.6.2 Member Function outside a Class
- 7.7 Inline Member Function
- 7.8 Array of Objects
- 7.9 Objects as Function Argument
  - 7.9.1 Pass by Value
  - 7.9.2 Pass by Reference
  - 7.9.3 Pass by Pointer
- 7.10 Friend Function and Friend Class
- 7.11 Static Data Member and Member Function
- 7.12 Let Us Sum Up
- 7.13 Answers to Check Your Progress
- 7.14 Further Readings
- 7.15 Model Questions

---

### 7.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to:

- learn about Class in C++
- identify basic components of a class
- declare a class and create objects
- define member function of a class



- describe array of Objects
- illustrate objects as function arguments

---

## 7.2 INTRODUCTION

---

So far, we have learnt that C++ lets you to create variables which can be of a range of basic data types : *int*, *long*, *double* and so on. However, the variables of the basic type do not allow you to model real –world objects (or even imaginary objects) adequately. We come to know that the basic theme of the object oriented approach is to model the real –world problems. So, object oriented programming language C++ introduces a new data type called **class** by which you can define your own data types as *class*. Defining the variables of a class data type is known as a class instantiation and such variables are called **objects**. In this unit, we will introduce you how to declare a class and how to create objects of a class. We will also discuss how a member function declare inside a class or outside a class and how it can be accessed. Moreover, the techniques of passing objects as function arguments are also illustrated in this unit.

---

## 7.3 CLASSES IN C++

---

We are already familiar with the term encapsulation which is a fundamental feature of OOP. The encapsulation is nothing but a mechanism that binds together the data and functions into a single component, and keeps both safe from outside interference and misuse.

The data cannot accessible by outside functions. With encapsulation data hiding can be accomplished.

In C++, the encapsulation is supported by a construct called- “*class*”. First, let us think a bit about-what is an object . From the general concept, we can say that an object is something that has fixed shape or well defined boundary. In other words, an object can be a person, a place, or a thing with which the computer must deal. If you look at your surroundings some

objects may correspond, to real-world entities such as– student, bank account, book, cars, bags, computer, lock, watch etc. You will observe two characteristics about objects–

firstly – each objects has certain attributes.

secondly – each objects has some behavioers or actions or operations associated with it.

For example, the object 'computer' & 'car' has the following attributes and operations–

Object : car

Attributes: company, model, colour, & capacity

Operation: speed ( ), average ( ), & break ( )

Object: Computer

Attributes: brand, price, monitor resolution, hard disk and RAM size

Operation: Processing( ), display( ) & printing ( )

Each object will have its own identity though its attributes and operation are some, the objects will never become equal. In cash of person object for instance, two person have the same attributes like name, age and sex, but they are not equal. Objects are the basic run time entities in an object – orieated system. Thus, in C++, an object is a collection of **related variables** and **function** bound together to form a high level entity.

The variable defines – the state of the object

and functions defines – the action or operation that can be performed on the object.

Now, let us come back to the discussion of class.

A class is a grouping of objects having identical attributes and common behaviour (operations). It means all objects possessing similar attributes or properties are grouped into the same unit, which is called a class. A class encloses both the data and function into a single unit as shown in the following fig.7.1.

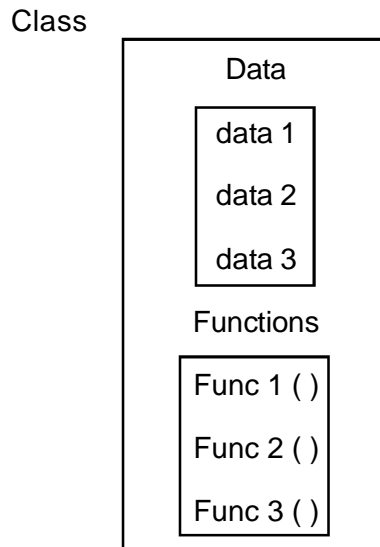


Fig. 7.1 Grouping of data and function in a class

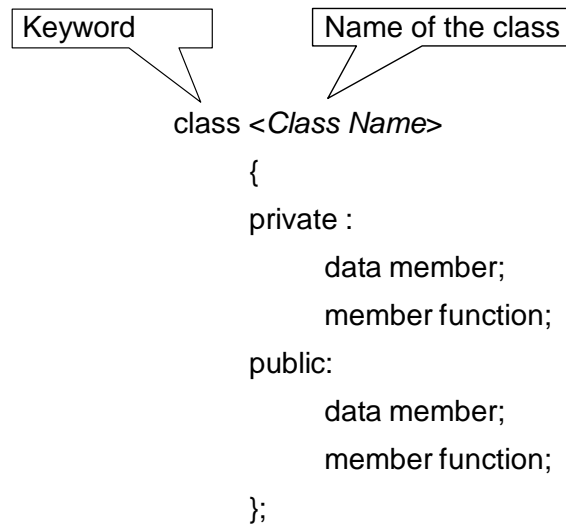
Thus, the entire group of data and code of an object can be built as a user-defined data type using class. It is obvious that classes, are the basic language construct in C++ for creating the user defined data types. Once a class have been declared, the programmer can create a number of objects associated with that class. Objects are nothing but the variable of the class data type. Defining variables of a class data type is known as a class instantiation. The syntax used to create an object of the class data type is similar to the syntax used to create an integer variable in C. In the next section, we will learn how to declare a class and an object.

---

## 7.4 CLASS DECLARATION

---

We come to know that – classes contain not only data but also functions. Data and functions within a class are called members of a class. The data inside a class is called a **data member** and the functions are called **member function**. The member functions define the set of operations that can be performed on the data member of a class. The syntax of a class declaration is shown below—



The keyword '**class**' indicates the name which follows class name, is an abstract data type. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The body of a class contains declaration of data members and member functions.

The members of a class are usually grouped in two sections i.e. *private* and *public*, which defines the visibility of members.

The following declaration illustrates a class specification:

```
class employee
{
 Private :
 char name[30];
 float age;
 Public :
 void insert_data (void);
 void show_data (void);
};
```

A class name should be meaningful, reflecting the information it holds. Here in our example, the class 'employee' contains two data members and two member functions. The member function *insert\_data()* is used to assign value to the member variable or data member 'name' and 'age'. The member function *show\_data()* is used to display the values of the data members.

The data member of the class `employee` cannot be accessed by any other functions that are defined outside the class. It means only the member functions of a class are permitted to access its data members.

In general, the data members are declared as private and member functions are declared as public. In our example, though we have not specify the data members as private yet they are treated as private by default.

The following figures represent the class 'employee'.

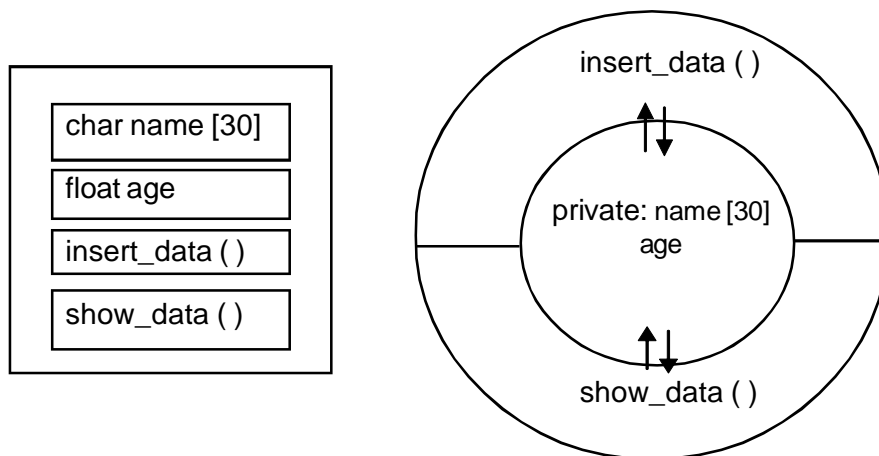


Fig. 7.2 Representation of 'employee' class

---

### 7.4.1 ACCESS CONTROL IN A CLASS

---

The members of a class are generally grouped into three sections by using the following keyword—

- private
- public
- protected

These keywords are called **access control specifiers**. These control specifiers are written inside the class and terminated by this ':' symbol. All the members that follow a keyword (upto another keyword) belong to that type. If no keyword is specified, then the members are assumed

as private member. We will discuss later about the protected keyword. Let us now briefly discuss about private and public keywords.

**Private:** Private members are accessible to their own class members only. They cannot be accessed from outside the class by any member functions. The members at the beginning of class without any access specifier are private by default. Hence, writing the keyword 'private' at the beginning of a class is optional.

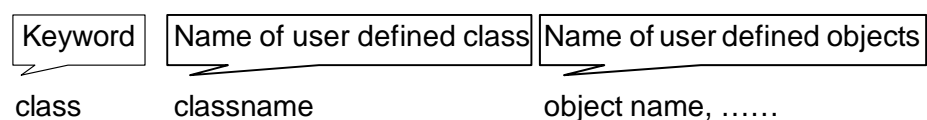
**Public:** Public members are visible (accessible) outside the class, they should be declared in public section. All data members are not only accessible to their own members of a class but also can be accessible from anywhere in the program, either by functions that belong to the class or by those external to the class.

## 7.5 DECLARING OBJECTS

A class declaration only builds the structure of objects. In our example, the class *employee* does not define any objects of employee but only specifies what it will contain. Once a class has been declared, we can create variable of that type by using the class name (like any other built-in type variable). The process of creating objects (variables) of the class is called **class instantiation**. For example—

```
int x, y, z; // declaration of integer variables
float m, n; // declaration of float variables
employee a; // declaration of object or class variable
```

The syntax for creating objects are shown below :



Remember, the use of the keyword 'class' is optional at the time creating objects.

For example, ***class employee e1;***  
or  
***employee e1;***

In a single statement we can create more than one objects as follows :

***employee e1, e2, e3, e4;***

Like in the case of structures, we can create objects by placing their names immediately just after the closing braces as follows –

```
class employee
{
 // body of the class
} e1, e2, e3;
```

This practice is rarely followed because we would like to define the objects as and when required, or at the point of their use.

Always remember that, at the time of declaration of object, necessary memory space is allocated for an object. Suppose, we have declared two objects as –

***employee e1, e2;***

Both **e1** and **e2** have the same data members and it is illustrated by the following figure.

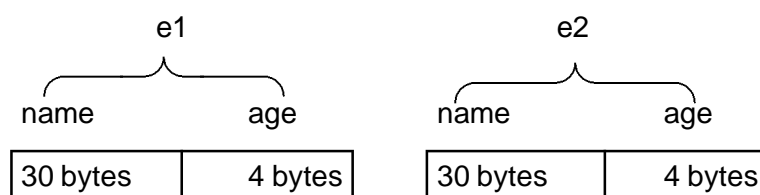


Fig. 7.3 Allocation of memory for objects

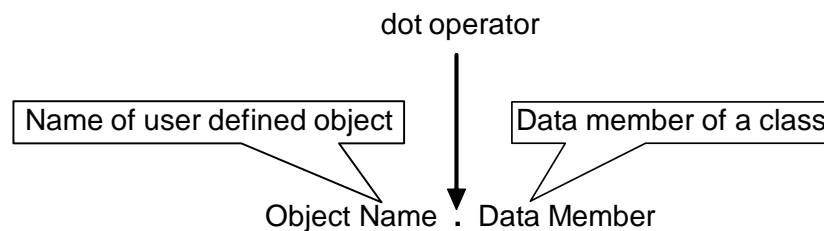
Here, in the figure the objects **e1** and **e2** occupies the memory area. They are not initialized to anything, however the data members of each object will simply contains junk values. So, our main task is to access the data members of the object and setting them to some specific values.

An object is a conceptual entity having the following properties:

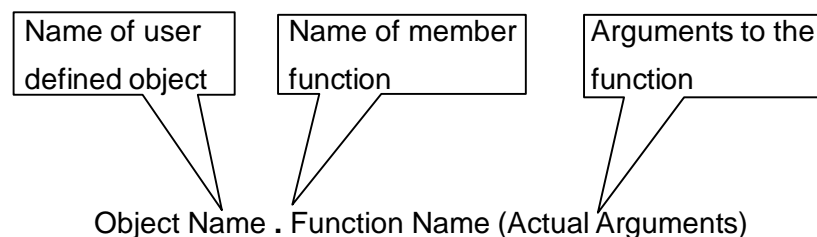
- it is individual
- it points to a thing, either physical or logical that is identifiable by the user.
- it holds data as well as operation method that handles data.
- its scope is limited to the block in which it is defined.

### 7.5.1 ACCESSING CLASS MEMBERS

After creating an object of a class, it is the time to access its members. This is achieved by using the member access operator, dot ( . ). The syntax for accessing members (data and functions) of a class is shown below—



(a) Accessing data member of a class



(b) Accessing member function of a class

The following program demonstrates how the objects are used for accessing the class data members.



**//Program 7.1**

```
include <iostream.h>
include <string.h>
include <conio.h>
class employee
{
 private:
 char name [30]; // name of an employee
 float age; // age of an employee
 public : // initializing data members
 void insert_data (char * name1, float age)
 {
 strcpy (name, name1);
 age = age1;
 }
 void show_data () //displaying the data members
 {
 cout << "Name :"<<name<<endl;
 cout << "Age :"<<age<<endl;
 }
};

void main ()
{
 employee e1; //first object of class employee
 employee e2; //second object of class employee
 clrscr();
 e1.insert_data ("Hemanga", 30); // e1 calls member insert_data ()
 e2.insert_data ("Prakash", 32); // e2 calls member insert_data ()
 cout << "Employee Details:"<<endl;
 e1.show_data (); // e2 calls member show_data ()
 e2.show_data (); // e2 calls member show_data ()
 getch ();
}
```

**RUN :** Employee Details:

Name : Hemanga  
Age : 30  
Name : Prakash  
Age : 32

In the above program, in `main()` we have declared two objects through the statements

**`employee e1;` and `employee e2;`**

The statements

`e1.insert_data ("Hemanga", 30);`  
`e2.insert_data ("Prakash", 32);`

initialize the data members of object `e1` and `e2`. The object `e1`'s data member 'name' is assigned 'Hemanga' and age is assigned 30. Similarly, the object `e2`'s data member 'name' is assigned 'Prakash' and age is assigned 32.

The statements

**`e1.show_data ();`**  
**`e2.show_data ();`**

call their member `show_data ()` to display the contents of data members namely, 'name' and 'age' of employee objects `e1` and `e2`. The two objects `e1` and `e2` will hold different data values.



### CHECK YOUR PROGRESS - 1

1. Answer the following by selecting the appropriate option:
  - a) The members of a class are by default.
    - (i) Private
    - (ii) Public
    - (iii) Protected
    - (iv) None of these
  - b) The private data of any class is accessed by -
    - (i) Only public member function
    - (ii) Only private member function
    - (iii) Boths (i) & (ii)
    - (iv) None of these

- c) Encapsulation means
- (i) Protecting data
  - (ii) Allowing global access
  - (iii) Data hiding
  - (iv) Both (i) & (iii)
- d) The size of object is equal to
- (i) Total size of member data variables
  - (ii) Total size of member functions
  - (iii) Both (i) & (ii)
  - (iv) None of these
- e) In a class, only member function can access data which is not accessible to out side. This feature is called
- (i) data security
  - (ii) data hiding
  - (iii) data manipulation
  - (iv) data definition

---

## 7.6 DEFINING MEMBER FUNCTIONS

---

We have already come to know that—a class holds both the data and functions which are called *data members and member functions*. The data member of a class must be declared within the body of the class. The member functions of a class can be defined in two place—

- o inside the class definition
- o outside the class definition

It is obvious that— a function should perform the same task, no matter where it is defined. But the syntax of a member function definition changes depending on the place of its definition, i.e. inside a class or outside a class. We will now discuss both the approaches.

---

### 7.6.1 MEMBER FUNCTION INSIDE A CLASS

---

In this method, the function is defined inside the class body. When a function is defined inside a class, it is treated as an **inline** function. We will discuss inline function in the next section.

In the program 9.1 we have defined the member functions–

```
void insert_data (char * name1, float age);
```

and

```
void show_data ();
```

inside the class 'employee'. We have seen that these function definition are similar to the normal function definition except that they are enclosed within the body of a class. They will treat as an inline function. Remember that if a function contains loop instruction i.e. *for*, *while do*, *while* ...etc. then that function will not treated as inline function.

---

### 7.6.2 MEMBER FUNCTION OUTSIDE A CLASS

---

In this method of defining a member function outside a class - first you will have to declare a function prototype within the body of the class and then define the function outside the body of the class.

The function defined outside the body of a class have the same syntax as normal functions i.e. they should have a function header and a function body. But, there must have a mechanism of binding the functions to the class to which they belong. This is done by using the **scope resolution operation (: :)**, in the header of the function. The scope resolution operator acts as an 'identity-label' and tells the compiler, the class to which the function belongs. The general form of a member function definition is shown below–

*ClassName*

```
{

 Return Type MemberFunction (arguments); // Function Prototype

};
```

*Return Type ClassName :: MemberFunction (arguments)*

```
{
 // body of the function
}
```

Here, the label **ClassName ::** tells the compiler that the function **MemberFunction** is the member of class **ClassName**. The scope of the function is restricted to only the objects and other members of the class. We can modify the Program 7.1 by defining the member functions outside the class body, as shown below:

### **//Program 7.2**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
 private:
 Char name [30];
 float age;
 public:
 void insert_data (char *name1, float age1);
 void show_data ();
};

void employee :: insert_data (char *name1, float age1)
// Function declaration
{
 strcpy (name, name1);
 age = age1;
}

void employee :: show_data () // Function definition
{
 cout <<"Name : "<<name<<endl;
 cout <<"Age: "<<age<<endl;
}
```

```
void main ()
{
 employee e1, e2;
 clrscr();
 e1 . insert_data ("Hemanga", 30);
 e2 . insert_data ("Prakash", 32);
 cout<<"EMPLOYEE DETAILS...."<<endl;
 e1 . show_data ();
 e2 . show_data ();
 getch ();
}
```

**RUN : EMPLOYEE DETAILS:**

```
Name : Hemanga
Age : 30
Name : Prakash
Age : 32
```

In the above definitions, the label '**employee :**' informs the compiler that the functions `insert_data ( )` and `show_data ( )` are the members of the employee class.

The member functions have some special characteristics that are—

- o A program can have several different classes and they can use same name for different member functions. The 'membership label' (ClassName : :) resolves the ambiguity of the compiler in deciding which function belong' to which class.
- o Member functions can access the private data of the class, whereas non-member functions are not allowed to access. But 'friend function' can access the private data member of a class we will discuss later about the friend functions.
- o Member functions of the same class can access all other members of their own class without the use of dot operator.

---

## 7.7 INLINE MEMBER FUNCTION

---

Let us first see what is an inline function. C++ provides a mechanism called *inline function*. When a function is declared as inline, function body is inserted in place of function call during compilation. In this mechanism, passing of control between *caller* and *callee* functions is avoided. The use of the inline function is most effective when calling function is small. If the calling function is very large, in such a case also, compiler copies the content of the inline function in called function which reduces the program execution speed. So, in such a case inline function should not be used.

Now, let us see how an inline function behave with class specification. We have come to know that we can define a member function outside the class definition. The same member function can be define as inline function by just using the qualifier inline in the header line of the function defining. In the followng, the syntax for defining inline function outside the class declaration is shown -

Keyword indicates function defined outside a class body is inline

***inline Return Type ClassName : : FunctionName (arguments)***

In fact, the inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is treated like a **macro** i.e. any call to this function in a program is replaced by the function itself. This is known as *inline expansion*. Inline functions are also called as open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function. By default, all member functions defined inside the class are inline function.

We can modify the program 7.1 by defining the member functions as inline function as shown below:

**// Program 7.3**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
 private:
 char name [30];
 float age;
 public :
 void insert_data (char *name1, float age1);
 void show_data ();
};

inline void employee :: insert_data (char *name1, float age)
{
 strcpy (name, name1);
 age = age1;
}

inline void employee :: show_data ()
{
 cout <<"Name :"<<name <<endl;
 cout <<"Age:" <<age <<endl;
}

void main ()
{
 employee e1, e2;
 clrscr();
 e1 . insert_data ("Hemanga", 30);
 e2 . insert_data ("Prakash", 32);
 cout << "Employee Details .." << endl;
 e1 . show_data ();
 e2 . show _data ();
 getch ();
}
```



**RUN** : Employee Details–

Name : Hemanga

Age : 30

Name : Prakash

Age : 32

---

## 7.8 ARRAY OF OBJECTS

---

We know that arrays holds data of similar type. Arrays can be of any data type including user defined data type, created by using *struct*, *class* etc. We can create an array of variables by using class data type. Such an array of variables of class data type is also known as an array of objects which handle a group of objects.

Let us consider the following class definition :

```
class employee
{
 private :
 char name [30]
 float age ;
 Public :
 void insert_data (char* name1, float age1);
 void show_data ();
};
```

Here, the identifier '*employee*' is a user defined data type and can be used to create objects that relate to different categories of employees. For example, the following definition will creates an array of objects of '*employee*' class–

```
employee consultant [30]; // array of consultant
employee clerk [15]; // array of clerk
employee lecturer [20]; // array of lecturer
```

The array consultant contain 30 objects (consultant), namely, consultant [0], consultant [1], .....consultant [29], of type employee class. Similarly, clerk array contains 15 objects and lecturer array contains 20 objects.

We know that, array elements occupies continuous memory locations like the same way array of objects occupies contiguous memory locations as shown in the following fig:

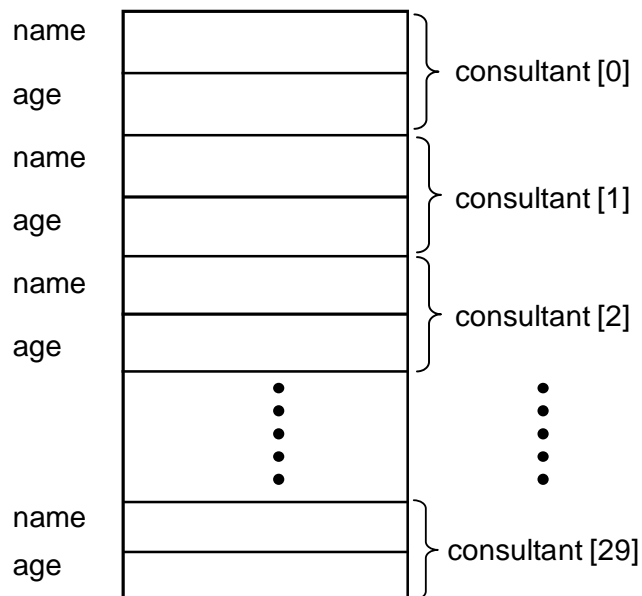


Fig.7.4 Storage of data items in 'consultant' array of objects.

By using index an individual element of an array of objects can be referred i.e. **consultant [15], consultant [9] ..... etc.** By using the dot operator [ . ] we can access any member of an object. For example

**consultant [ 30 ] . show\_data ( )**

will display the data of 30th consultant.

We can rewrite the program 7.1 by using array of objects as shown below:

#### // Program 7.4

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
```

```
{
 private:
 char name [30];
 float age;
 public:
 void insert_data (char *name1, float age)
 {
 strcpy (name, name1);
 age = age1;
 }
 void show_data ()
 {
 cout<<"Name:"<<name<<endl;
 cout<<"Age:"<<age<<endl;
 }
};

void main ()
{
 int i, age, count;
 char name [30], tag;
 employee consultant [30];
 clrscr ();
 count = 0;
 for (i=0; i<30; i++)
 {
 cout<<"Enter Data For Employee (Y/N):";
 cin >> tag;
 if (tag == 'y' || tag == 'Y')
 {
 cout <<"\n Enter Name of Employee:";
 cin >> name;
 cout << "Age:";
 cin >> age;
 consultant [i] . insert_data (name, age);
 count ++;
 }
 }
}
```

```

 }
 else
 break;
 }

 cout << "\n\n Employee Details \n" ;
 for (i=0; i < count; i++)
 consultant [i] . show_data ();
 getch ();
}

```

**RUN :** Enter Data for Employee (Y/N) : y  
 Enter Name of Employee : Prakash  
 Age : 30  
 Enter Data for Employee (Y/N) : y  
 Enter Name of Employee : Hemanga  
 Age : 28  
 Enter Data for Employee (Y/N) : n  
 Employee Details...  
 Name : Prakash  
 Age : 30  
 Name : Hemanga  
 Age : 28

---

## 7.9 OBJECTS AS FUNCTION ARGUMENTS

---

Objects can be passed as an argument to a function. There are three ways of passing objects as function arguments:

- o a copy of the entire object is passed to the function, which is also called pass-by-value
- o only the address of the object is sent implicitly to the function, which is also called – pass by-reference.
- o the address of the object is sent explicitly to the function, which is also called – pass-by-pointer.

---

## 7.9.1 PASS-BY-VALUE

---

In this technique, a copy of the object is passed to the called function (callee) from the calling function (caller). Since a copy of the object is passed so any changes made to the object inside the called function do not affect in the object used to call the function.

The following program demonstrates the use of objects as function arguments in pass-by-value mechanism.

### //Program 7.5

```
#include<iostream.h>
#include<conio.h>
class age
{
 private:
 int birthyr ;
 int presentyr ;
 int year ;
 public:
 void getdata ();
 void period (age);
};

void age :: getdata ()
{
 cout<<" \n Year of Birth:";
 cin >> birthyr ;
 cout << "Current year:" ;
 cin >> presentyr ;
}

void age :: period (age x1)
{
 year = x1 . presentyr - x1 . birthyr ;
 cout << "Your Present Age :" <<year<<"Years" ;
}
```

```
void main ()
{
 clrscr ();
 age a1 ;
 a1 . getdata ();
 a1 . period (a1) ;
 getch ();
}
```

**RUN:** Year of Birth : 1990  
Current Year : 2002  
Your Present Age : 19 years

In the above program, the class age has three data member. The function *getdata* ( ) reads integers through keyboard. The function *period* ( ) calculates the difference between the two integers. In function main ( ), a1 is an object to the class age. The object a1 calls the function *getdata* ( ). The same object a1 is passed to the function *period* ( ), which calculates the difference between the two integers. Thus, an object can be passed to a function.

---

## 7.9.2 PASS-BY-REFERENCE

---

In this technique, only the address of the object is sent to the function. When an address of the object is passed, the address acts as reference pointer to the actual object in the calling function. Therefore, any change made to the objects inside the called function will reflect in the actual object in the calling function. We can modify the program 7.5 by using the pass by reference mechanism -

### // Program 7.6

```
#include<iostream.h>
#include<conio.h>
class age
```

```
{
 private:
 int birthyr ;
 int presentyr ;
 int year ;

 public:
 void getdata ();
 void period (age);
};

void age :: getdata ()
{
 cout<<" Year of Birth:";
 cin >> birthyr ;
 cout << "Current year:" ;
 cin >> presentyr ;
}

void age :: period (age & x1)
{
 x1. year = x1 . presentyr - x1 . birthyr ;
 cout << "Your Present Age :" <<year<<"Years" ;
}

void main ()
{
 clrscr ();
 age a1 ;
 a1 . getdata ();
 a1 . period (a1) ;
 getch ();
}
```

**RUN :** Year of Birth : 1990  
Current Year : 2009  
Your Present Age : 19 years

---

### 7.9.3 PASS-BY-POINTER

---

In this mechanism also, the address of the object is passed explicitly to the called function from the calling function. The program 7.6 is modified by using the mechanism pass-by-pointer as follows:

**//Program 7.7**

```
#include<iostream.h>
#include<conio.h>
class age
{
 private:
 int birthyr ;
 int presentyr ;
 int year ;
 public:
 void getdata ();
 void period (age *);
};

void age :: getdata ()
{
 cout<<" \n Year of Birth:";
 cin >> birthyr ;
 cout << "current year:" ;
 cin >> presentyr ;
}

void age :: period (age * x1)
{
 year = x1 → presentyr - x1 → birthyr ;
 cout << "Your Present Age :" <<year<<"Years" ;
}

void main ()
{
 clrscr ();
 age a1 ;
```



```

 a1 . getdata ();
 a1 . period (&a1) ;
 getch ();
 }

```

```

RUN: Year of Birth : 1990
 Current Year : 2009
 Your Present Age : 19 years

```

In the programs 7.6 & 7.7 you just keep an eye on the symbols '.', '→', '\*', '&', and the statements(bold lines) where we have appropriately used them.

---

## 7.10 FRIEND FUNCTION AND FRIEND CLASS

---

We have already discussed about the fact that the private members of a class cannot be accessible from the outside of the class, only the member functions of that class have permission for accessing the private members. This policy enforces the encapsulation and data hiding techniques.

Let us think about a situation where a user needs a function to operate on objects of two different classes. It means, that function will be allowed to access the private data of both the classes. In C++, this situation is overcome by using the concept of friend function. It permits a friend function to access the different class's private members.

The declaration of a friend function must be prefixed by the keyword "**friend**". In the following class, shows a declaration of a friend function.

```

class test
{
 private:

 public :

```

```


 friend void sum() ;
};

```

The function can be defined any where in the program similar to any normal C++ function. The function definition does not use either the keyword friend or the scope operator ': :'. The functions that are declared with the keyword 'friend' are called friend functions. A friend function can be a friend to a multiple classes. The friend function have the following properties :

- o There is no scope restriction for the friend function; hence they can be called directly without using objects.
- o Unlike member functions of class, friend function cannot access the members directly. On the other hand, it uses object and dot operator to access the private and public member variables of the class.
- o Use of friend function is rarely done, because it violates the rule of encapsulation and data hiding.
- o The function can be declared in public or private sections without changing its meaning.

The following program demonstrates the use of friend function:

### **// Program 7.8**

```

#include<iostream.h>
#include<conio.h>
class first ; /*forward declaration like function Prototype*/
class second
{
 int x ;
 public :
 void get value ()
 {
 cout << "\n Enter a number :";
 cin >> x ;
 }
}

```

```
 friend void sum (second , first) ; // declaration of friend dunction
 };
class first
{
 int y ;
 public :
 void getvalue ()
 {
 cout << "\n Enter a number:" ;
 cin >> y ;
 }
 friend void sum (second, first) ;
};

void sum (second m, first n)
{
 cout << "\n Sum of two numbers :." << n.y + m.x
}

void main()
{
 clrscr () ;
 first a ;
 second b ;
 a. get value () ;
 b. get value () ;
 sum (b, a), //funciton is called like a general function in C++
}
```

**RUN :** Enter a number : 9  
Enter a number : 12  
Sum of two numbers : 21

In the above program each of the two classes 'first' and 'second' has a member function named getvalue ( ) and one private data member. Notice that, the function sum ( ) is declared as friend function in both the class. Hence, this function has the ability to access the members of both the classes. Using sum ( ) function, addition of integers is calculated and displayed.

It is possible to declare all the member functions of a class as the friend functions of another class. When all the functions need to access another class in such a situation we can declare an entire class as **friend class**. Always remember friendship is not exchangeable, its meaning is that - declaring *class A* to be a friend of *class B* does not mean that *class B* is also a friend of *class A*. The declaration of a friend class as follows:

```
class second
{

 friend class first;
}; /* all member functions of class first are friends to
 class second */
```

The following program demonstrates the use of friend class :

#### **//Program 7.9**

```
#include<iostream.h>
#include<conio.h>
class smallvalue;
class value
{
 int a;
 int b;
 public:
 value (int i, int j) // declaration of constructor with arguments
 {
 a = i;
 b = j;
 }

 friend class smallvalue;
};
class smallvalue
{
 public:
```

```
int minimum(value x)
{
 return x.a < x.b ? x.a : x.b;
}

};

void main ()
{
 value x (15, 25);
 clrscr () ;
 smallvalue y;
 cout << y. minimum(x);
 getch ();
}
```

In the above program we have used the constructor with arguments. The concept of constructors are illustrated in unit 8 'Constructors and Destructors'.



## CHECK YOUR PROGRESS - 2

1. Fill in the blanks of the following :
  - (i) Member functions defined within the class definition are implicitly \_\_\_\_.
  - (ii) When only the address of the object is sent explicitly, it is called \_\_\_\_.
  - (iii) \_\_\_\_ function can access the private data members of a class.
2. State whether the following statements are true or false:
  - (a) To reference an object using a pointer to object, uses the<> operator.
  - (b) In the prototype void sum (int &) arguments are passed by reference.
  - (c) If class A is a friend class of class B then a member function of class B can access the data members of class A.

## 7.11 STATIC DATA MEMBER AND MEMBER FUNCTION

After studying public and private members, let us study about the static members of a class. Recall what we have learn from C-Programming:

- (i) A variable can be declared as static inside a function or outside main( ).
- (ii) Static variables value do not disappear when function is no longer active, their last updated value always persist. That is when the control come back to the same function again the static variables have the same value as they leave at the last time.

in C++ also. However, C++ has objects. Hence, the meaning of static with respect to member variables of an object is different.

We have already gained the idea is that each object has its separate set of data member variable in memory. The member functions are created only once and all object share the function. No separate copy of function of each object is created in the memory like data member variables. The following fig. shows the accessing of member function by objects.

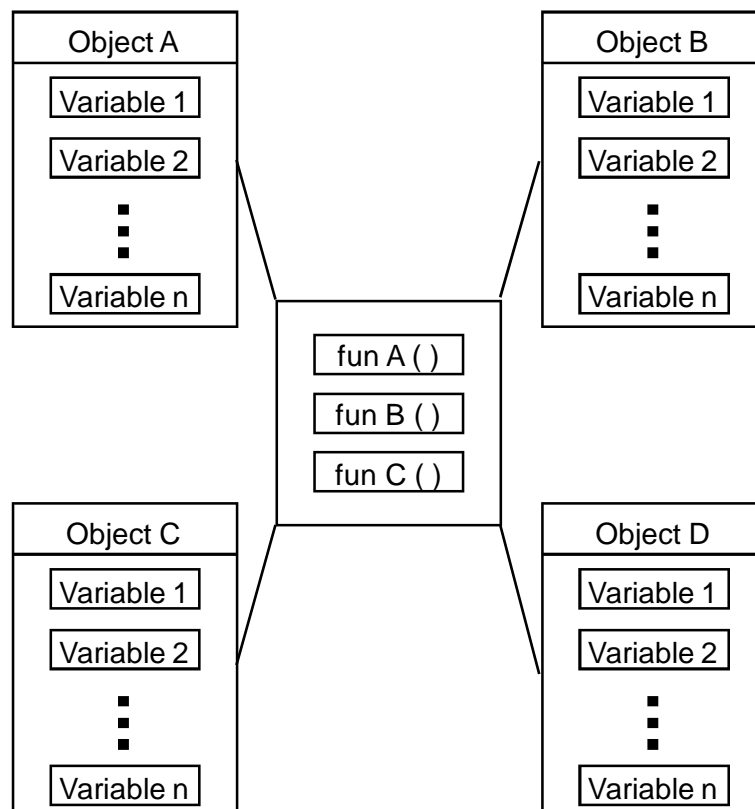


Fig. 7.5 Data members and member functions in memory

In C++, it is possible to create common member variables like function using the **static** keyword. Once a data member variable is declared as static, only one copy of the member is created for the whole class and that all objects of the class will share that variable.

Always remember–

- o A static variable preserve the value of a variable.
- o When a variable is declared as static it is initialize to zero.
- o A static data member or member function is only recognized inside the scope of the present class.
- o A static variable can be a public or private.

The syntax for declaring static data member or member function within a class is shown below:

***static <variable name> ;***

***static <function name> ;***

When you declare a static data member within a class, you are not defining it i.e. you are not allocating storage for it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

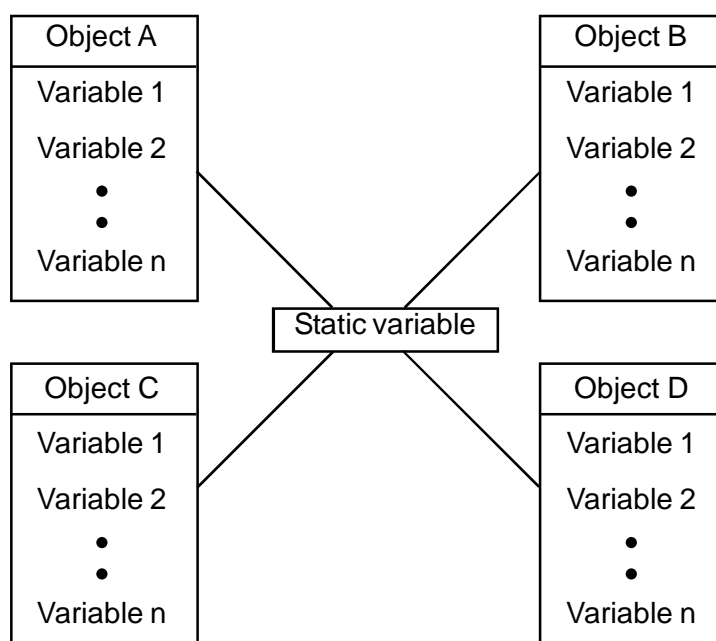


Fig. 7.6 Static member in memory

The declaration of static member is shown below :

```
class number
{
 static int C;
 public:

};

int number : : C = 0 // initializaiton of static member variable
```

The following program demonstrates the use of static data member in a class—

**// Program 7.10**

```
#include<stdio.h>
#include<conio.h>
class number
{
 static int C;
 public:
 void count ()
 {
 C ++;
 cout << "\n C =" << C;
 }
};

int number : : C = 0;

void main ()
{
 number a, b, c;
 clrscr () ;
 a.count ();
 b.count ();
 c.count ();
 getch ();
```



```
 }
RUN : c = 1
 c = 2
 c = 3
```

In the above program, the class `number` has one static data variable `C`. The `count()` is a member function, increment value of static member variable `C` by 1 when called. The statement `int number : : C = 0` initialize the static member with 0. It is possible to initialize the static data members with other values. In the function `main()`, `a`, `b` and `c` are three objects of `class number`. Each object calls the function `count()`. At each call to the function `count()` the variable `C` gets incremented and the count statement displays the value of variable `C`. The objects `a`, `b` and `c` share the same copy of static data member `C`.

### STATIC MEMBER FUNCTION

In C++, like member variables, functions can also be declared as static. When a function is defined as static, it can access only static member variable and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The static keyword makes the function free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program.

The following points should be remembered while declaring static function:

- a) Just one copy of the static member is created in the memory for entire class. All objects of the class share the same copy of static member.
- b) Static member functions can access only static data member, or functions.
- c) Static member functions can be invoked using class name.
- d) It is also possible to invoke static member functions using objects.
- e) When one of the objects changes the value of data member variables,

the effect is visible to all the objects of the class.

The following program demonstrates the use of the static member function in a class.

**//Program 7.11**

```
#include<iostream.h>
#include<conio.h>
class number
{
 private:
 static int X;
 public:
 static void count () {X++; }
 static void display ()
 {
 cout << "\n value of X =" << X;
 }
};

int number : : X = 0;
void main ()
{
 clrscr () ;
 number : : display () ; //invokes display function
 number : : count () ; //invokes count function
 number : : count () ; //invokes display function
 number : : display () ; //invokes display function
 getch () ;
}
```

**RUN:** Value of X : 0

Value of X : 2

In the above program, the member variable X and functions *count ( )* & *display ( )* of *class number* are static. The function *count ( )* when called, increases the value of static variable X. The function *display ( )* prints the current value of the variable X. The static functions can be called using class name and scope resolution operator as shown in the program–

***number : : count ( ) ;***

---

***number :: display ( );***

---

## **9.12 LET US SUM UP**

---

- i) Classes are the basic language construct in C++ for creating the user defined data types.
- ii) A class contains member variable or data members and member functions.
- iii) The members of a class are grouped into two sections namely private and public.
- iv) Defining variables of a class data type is known as a class instantiation and such variables are called objects.
- v) Using the member access operator, dot(.), the class members can be access by the objects.
- vi) The member function can be defined as a) private or public b) inside the class or outside the class.
- vii) The scope resolution operator (::) is used, when a member function is defined outside the class body.
- viii) Inline member function is treated like a macro, when a function is declared as inline, function body is inserted in place of function call during compilation.
- ix) We can create an array of variables by using the class data type, then these variables are called array of objects, which occupies contiguous memory locations in memory.
- x) There are three methods of passing objects to function, namely, pass-by-value, pass-by-reference and pass-by-pointer.
- xi) The function that are declared with the keyword ***friend*** are called friend function. A function can be a friend to multiple classes.
- xii) static is the keyword used to preserved value of a variable. When a variable is declared as static, it is initialize to zero. A static function or data element is only recognized inside the scope of the present class.
- xiii) When a function is defined as static, it can access only static member variables and functions of the same class. The static member

functions are called using its class name without using its objects.



## 7.13 ANSWERS TO CHECK YOUR PROGRESS

### CHECK YOUR PROGRESS - 1

1. a) i.                      b) iii.                      c) iv.                      d) i.                      e) ii

### CHECK YOUR PROGRESS - 2

1. i) inline,                      ii) pass-by-pointer,                      iii) friend  
2. a) False,                      b) True,                      c) False



## 7.14 FURTHER READINGS

1. Object Oriented Programming with C++, E. Balagurusamy, Tata McGraw Hill Publication.
2. The Complete Reference C++, Herbert Schildt, Tata McGraw Hill Publication.
3. Mastering C++, K.R. Venugopal, Rajkumar, T. Ravi Shankar, Tata McGraw Hill Publication.



## 7.15 MODEL QUESTIONS

1. What is a class ? How does it accomplish data hiding ?
2. What is an object ? How are they created ?
3. How a member function of a class is defined or declared ?
4. Explain the use of *private* and *public* keywords. How are they different from each other ?
5. What is the significance of scope resolution operator :: ?
6. When will you make a function inline and why ?
7. Explain the different methods of passing objects to functions.
8. What is a friend function and a friend class ? Explain with example.

List out the merits and demerits of using a friend function.

9. What is a static member function ? When do we declare a member of a class static ?
10. Define a class **student** with the following specifications :

**private members of the class :**

*admno*                      *integer*  
*sname*                      *20 character*  
*eng, mamth, science*   *float*  
*total*                      *float*  
*ctotal( )*                  *A function to calculate eng+math+science*  
                                  *with float return type*

**public member functions of class student :**

*Takedata( )*    *–function to accept values for admno, sname, eng, math, science and invoke ctotal( ) to calculate total.*

*Showdata( )*    *–function to display all the data members on the screen*

11. Define a class BOOK with the following specification :

**private members of the class BOOK are :**

*BOOK\_NO*                *integer type*  
*BOOK\_TITLE*          *20 characters*  
*PRICE*                   *float (price per copy)*  
*TOTAL\_COST( )*        *A function to calculate the total cost for N number of copies, where N is passed to the function as argument*

**public member function of the class BOOK are :**

*INPUT( )*                *function to read BOOK\_NO, BOOK\_TITLE, PRICE*

*PURCHASE( )*        *function to ask the user to input the number of copies to be purchased. It invokes TOTAL\_COST( ) and prints the total cost to be paid by the user.*

12. Define a class employee with the following specifications :

**private members of the class :**

*EMPNO*                   *integer*  
*ENAME*                  *20 character*  
*BASIC, HRA, DA*       *float*

*NETPAY*                      *float*

*calculate( )*                *A function to find BASIC+HRA+DA with float  
return type*

***public member of the class :***

*havedata( )*    *function to accept values for EMPNO, ENAME,  
BASIC, HRA, DA and invoke calculate( ) to calculate  
NETPAY*

*dispdata( )*    *function to display all the data members on the screen*

13. A class student has three data members : name, roll number, marks of 5 subjects and member function to assign streams on the basis of table given below :

**Average Marks Stream**

|             |                  |
|-------------|------------------|
| 96% or more | Computer Science |
| 91% - 95%   | Electronics      |
| 86% - 90%   | Mechanical       |
| 81% - 85%   | Electrical       |
| 76% - 80%   | Chemical         |
| 71% - 75%   | Civil            |

Declare the class student and define the member function.

---

## Unit 8: CONSTRUCTOR AND DESTRUCTOR

---

### UNIT STRUCTURE

8.1 Unit Introduction

8.2 Unit Objectives

8.3 Constructor

8.3.1 Default Constructor

8.3.2 Parameterized Constructor

8.3.3 Copy Constructor

8.4 Destructor

8.5 Dynamic Initialization of Objects

8.6 Summary

8.7 Key Terms

8.8 Model Questions

8.9 Further Reading

---

### 8.1 UNIT INTRODUCTION

---

Now we come to a very important and interesting part of C++ programming. As in the last unit we got some good understanding of Objects in C++ programming language, in this unit we will learn some important techniques of initialization of a class variable as a whole which are called **Constructor**. Similarly we will also learn process of releasing a class variable, which is called **Destructor**. In this unit, we will also learn Dynamic initialization of objects.

---

### 8.2 UNIT OBJECTIVES

---

After going through this unit, you will be able to:

- Explain the need of a Constructor
- Define and call a Constructor
- Understand the different type of Constructors

- Explain the various features and uses of Destructor

---

## 8.3 CONSTRUCTOR

---

A Constructor is a special purpose member function of a Class in C++ that allows a programmer to initialize the data member of an Object. Constructors are called special because of the following reasons

- (a) their names are same as of the Class.
- (b) A Constructor is automatically invoked whenever an object of a Class is created.
- (c) unlike other member function it does not have any return type, on declaring and defining a Constructor we does not even need to write the keyword *void*.
- (d) a constructor must be declared inside the public section
- (e) a constructor cannot be inherited.

Bellow you can find a simple example of a Constructor

---

### EXAMPLE 8.1

---

```
class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
roll=0;
cls=0;
name=new char(10);
strcpy(name,"");
}
 void enterdata(){
 cout<<"\nEnter your Name=";
 cin>>name;
 cout<<"\nEnter Class=";
 cin>>cls;
 cout<<"\nEnter Roll=";
 cin>>roll;
 }
}
```



```

 void getdata(){
 cout<<"\n.....The Output.....\n\n";

 cout<<"\nName="<<name<<"\nClass="<<cls<<"\nRoll
 ="<<roll;
 }

};
void main()
{
 student s1;
 s1.enterdata();
 s1.getdata();
}

```

In the figure above a constructor of student class is defined and declared; as we can see from the example the name of the constructor is same as the class. When ever an object of student Class is created it automatically invokes the **student ()** constructor in the public section of the class. As you can see from this example there is no extra step required to invoke the constructor. The constructor does not have any return type, even the void is also missing.

Please try to understand the steps mentioned in the example, in the remaining part of the unit we will try to build-up example based on this.

---

### 8.3.1. DEFAULT CONSTRUCTOR

---

**A Default Constructor is the one that has no parameter.** It is not mandatory to define a default Constructor, whenever an object of a class is declared having no parameter, the default constructor will automatically invoked as shown in the example 8.1. But if the programmer forgot to define a default constructor in his programme, the compiler implicitly declare a default constructor, which will initialize the data member of the class with some garbage value.

If we look at the example 8.1, we can get a good understanding of a default constructor. Here when the object s1 of the student class is created, the compiler automatically invoke the default constructor student (), where different initialization and memory allocation task are carried out.

---

### 8.3.2 PARAMETERIZED CONSTRUCTOR

---

When we need to initialize the data members of different object of a class with distinct value we need to pass the value or values to the constructors. A parameterized constructor is a constructor that accepts one or more parameter at the time of declaration of the objects.

Let us take the following example

---

#### EXAMPLE 8.2: TO ILLUSTRATE THE PARAMETERIZED CONSTRUCTOR

---

```
#include<iostream.h>
#include<string.h>
class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
 roll=0;
 cls=0;
 name=new char(10);
 strcpy(name,"");
 }
 student(int r, int c, char *n){
 roll=r;
 cls=c;
 name=new char(10);
 strcpy(name,n);
 }
}
```

```

 void enterdata(){
 cout<<"\nEnter your Name=";
 cin>>name;
 cout<<"\nEnter Class=";
 cin>>cls;
 cout<<"\nEnter Roll=";
 cin>>roll;
 }
 void getdata(){

 cout<<"\nName="<<name<<"\nClass="<<cls<<"\nRoll="<<roll<<"\n\n\n\n";
 }

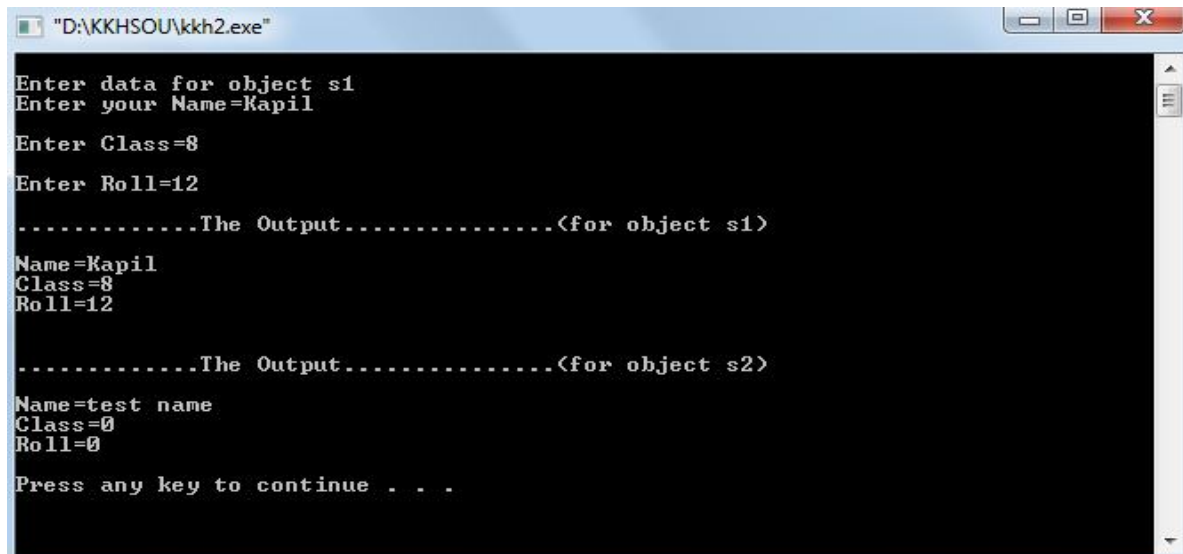
};

int main()
{
 student s1, s2(0,0,"test name");
 //Enter data for s1 object
 cout<<"\nEnter data for object s1";
 s1.enterdata();
 cout<<"\n.....The Output.....(for object s1)\n\n";
 s1.getdata();
 cout<<"\n.....The Output.....(for object s2)\n\n";
 s2.getdata();
 return 1;
}

```

---

The output of the above program



```
"D:\KKHSOU\kkh2.exe"

Enter data for object s1
Enter your Name=Kapil

Enter Class=8

Enter Roll=12

.....The Output.....<for object s1>

Name=Kapil
Class=8
Roll=12

.....The Output.....<for object s2>

Name=test name
Class=0
Roll=0

Press any key to continue . . .
```

In the program above we have tried to explain a parameterized constructor, if you look at the statement **s2(0,0,"test name")** at the main section of the program, you will get a good understanding of parameterized constructor. The Example 8.2 is an extension of the Example 8.1. Here when an object s2 is created, the constructor definition *student(int r, int c, char \*n)* is invoked.

In this program we have populated the data members of the object s1 using the function *enterdata()*, where as for the object s2, we have used the parameterized constructor.

If we use a *enterdata()* function for s2 object the initial data of the data members of object s2 will be overwrite by the new data entered by the user.

There are two ways to declare a parameterized constructor

- By calling the constructor implicitly (the method used above)
- By calling the constructor explicitly

For the first method **By calling the constructor implicitly** the following syntax is used

```
student s2(0,0,"test name");
```

Here we have declared an object of a Class s2 with appropriate values for the data members for the Class student passed from it. This is a more popular method and used by the majority of the programmers.

In the second method i.e. **By calling the constructor explicitly** a different syntax is used

```
student s2=student(0,0,"test name");
```

The results of both the methods are same.

---

### 8.3.3. COPY CONSTRUCTOR

---

The C++ compiler gives us lots of facilities for smooth completion of our different time consuming tasks. The one of the facility is the Copy Constructor, which allows us to copy the values of the data member of one object to the other object of Class using simple assignment operators. Below we will go to see a small example program for Copy Constructor

---

#### EXAMPLE 8.3: COPY CONSTRUCTOR

---

```
#include<iostream.h>
#include<string.h>
class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
 roll=0;
 cls=0;
 name=new char(10);
 strcpy(name,"");
 }
}
```

```
 student(int r, int c, char *n){
 roll=r;
 cls=c;
 name=new char(10);
 strcpy(name,n);

 }

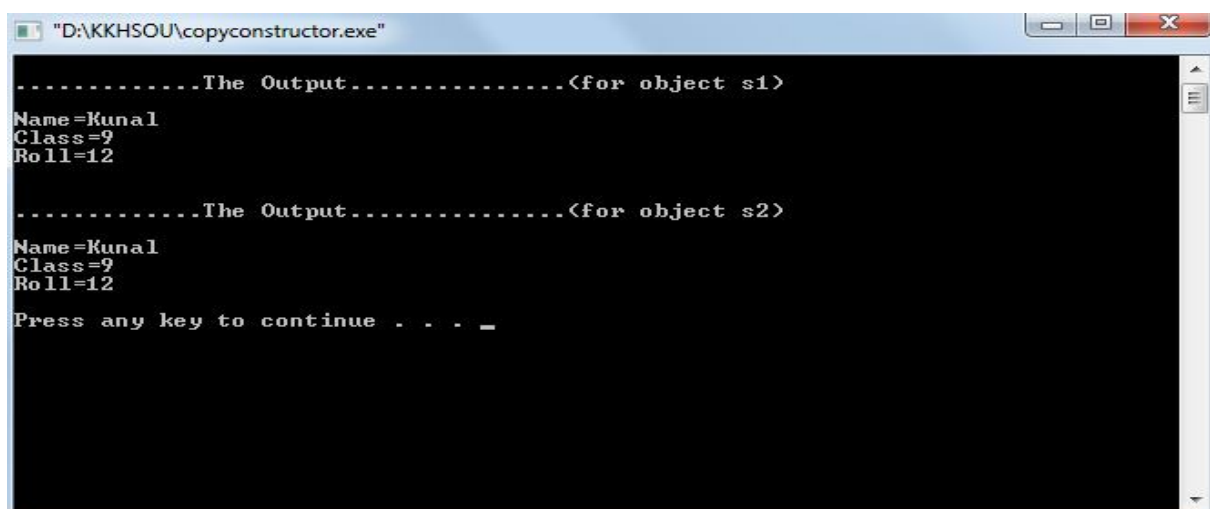
 void getdata(){

 cout<<"\nName="<<name<<"\nClass="<<cls<<"\nRoll
 ="<<roll<<"\n\n";
 }

};
int main()
{
 student s1, s2(12,9,"Kunal");
 //Copy Constructor
 s1=s2;
 cout<<"\n.....The Output.....(for object s1)\n";
 s1.getdata();
 cout<<"\n.....The Output.....(for object s2)\n";
 s2.getdata();
 return 1;
}
```

---

The output of the above program



```
"D:\KKHSOU\copyconstructor.exe"

.....The Output.....<for object s1>
Name=Kunal
Class=9
Roll=12

.....The Output.....<for object s2>
Name=Kunal
Class=9
Roll=12
Press any key to continue . . . _
```

In the Example 8.3 we have illustrated a small program for Copy Constructor. If you put your eyes at the first two lines of the main section of the program, you will see a different type of syntax. The syntax of the first line is same as that we have used in our previous examples, but the second line is bit different. Here we have tried to assign an object to another object using simple *assignment operator*. This is called Copy Constructor (or *Default Copy Constructor*). Here we have not write any controlling statement for **s1=s2**, the all the operations of copying the value of the data members of object s2 to object s1 are carried out by the Default Copy Constructor provided by the *Compiler* itself. From the output of the Example 8.3 given above we can see the successful completion of the step s1=s2. A Copy Constructor can be called using the syntax too s1(s2).

### **A Customized Copy Constructor handler**

A programmer can also write his own handler for a Copy Constructor, where he can provide his own customized code. When a customized Copy Constructor is written it will replace the Default handler provided by the Compiler. Bellow we will going to provide an example of the programmer defined handler for Copy Constructor.

---

#### **EXAMPLE 8.4: COPY CONSTRUCTOR**

---

```
#include<iostream.h>
#include<string.h>
class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
 roll=0;
 cls=0;
 name=new char(10);
```

```

 strcpy(name,"");
 }
 student(int r, int c, char *n){
 roll=r;
 cls=c;
 name=new char(10);
 strcpy(name,n);
 }
 student operator=(student &s){

 cout<<"\nUserdefined code for Copy
Constructor handler";
 roll=s.roll;
 cls=s.cls;
 strcpy(name,s.name);
 }

 void getdata(){

 cout<<"\nName="<<name<<"\nClass="<<cls<<"\nRoll
="<<roll<<"\n\n";
 }

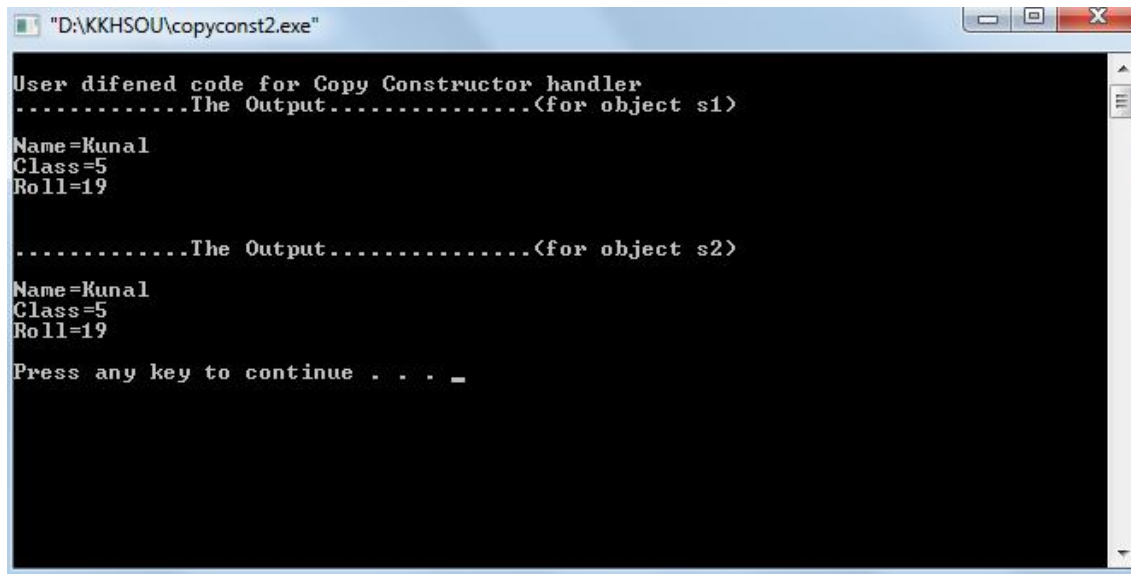
};
int main()
{
 student s1, s2(19,5,"Kunal");
 //Copy Constructor
 s1=s2;
 cout<<"\n.....The Output.....(for object s1)\n";
 s1.getdata();
 cout<<"\n.....The Output.....(for object s2)\n";
 s2.getdata();
 return 1;
}

```

---

The output of the above program





```
"D:\KKHSOU\copyconst2.exe"
User difened code for Copy Constructor handler
.....The Output.....<for object s1>
Name=Kunal
Class=5
Roll=19
.....The Output.....<for object s2>
Name=Kunal
Class=5
Roll=19
Press any key to continue . . . _
```

If you go through the above example, you will get everything familiar, except the function ***student operator=(student &s)***; this is the customized handler for the Copy Constructor. This is an operator function and this type of syntax is often used to overload a operator (you will get details of this in the next unit).

When the compiler found the statement `s1=s2`, it invokes the function “`student operator=(student &s)`”. In this function a local object is created by the compiler, here comes a small problem of wastage of memory. Though in this program it will not going to create any difficulties, but in case of a larger program having lots of data member in the Class, will waste lots of memory.

In the function `student operator=(student &s)`, the object `s` is the reference to object `s2` in the main section of the program. If you go through the steps in the function you will find that each and every data member of the `s2` object is assigned to the temporary object's data member, and in the end the temporary object returns to the main function implicitly and assigned to the object `s1`. This is the simple mechanism of a Copy Constructor handler.

**CHECK YOUR PROGRESS**

- What do you mean by a constructor in C++?
- Write down different use of a constructor.
- Explain different type of constructors.
- What do you mean by a “Copy Constructor”?

---

**8.4: DESTRUCTOR:**

A destructor, as the name suggested it is used for destroying an object. By destroying we mean deleting the data in the data members and releasing the memory space occupied by the data members. A destructor is also a member function of the class, like a constructor a destructor has the same name as that of the Class and it is also defined in the public section of the Class. For example if we want to write a destructor of the Class student we will define it as follows

```
~student(){ }
```

A destructor never takes any argument nor does it return any value. It will be implicitly invoked by the compiler upon exit from the function to clean up the storage that is no longer accessible.

---

**EXAMPLE 8.5: SYNTAX OF DESTRUCTOR**

```
class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
 roll=0;
 cls=0;
 name=new char(10);
 strcpy(name, "");
 }
}
```

```

void enterdata() {
 cout<<"\nEnter your Name=";
 cin>>name;
 cout<<"\nEnter Class=";
 cin>>cls;
 cout<<"\nEnter Roll=";
 cin>>roll;
}
void getdata() {
 cout<<"\n.....The
 Output.....\n\n";
 cout<<"\nName="<<name<<"\nClass="<<cls<<
"\nRoll="<<roll;
}

~student() {
 Cout<<"Object Destroyed.....\n";
}

};
void main()
{
 student s1;
 s1.enterdata();
 s1.getdata();
}

```

As we can see from the example above, as Constructor a Destructor also invoked implicitly.



### CHECK YOUR PROGRESS

- What do you mean by Destructor in C++?
- How do we can invoke a Destructor in C++?
- Is it mandatory to define a Destructor in C++ ? Justify your answer.
- Write a program to demonstrate the syntax and use of a Destructor

---

## 8.5 DYNAMIC INITIALIZATION OF OBJECTS

---

The concept of initializing an Object dynamically (at the run time not in the compile time) is very simple but it gives a good amount of flexibility to the programmer. It allows a programmer to initialize the data member of an object at the run time, so the initial value of an object can be changed depending the state of an program. Bellow we are going to demonstrate an example of the Dynamic Initialization of Object.

---

### EXAMPLE 8.5 DYNAMIC INITIALIZATION OF OBJECT

---

```
#include<iostream.h>
#include<string.h>

class student{
private:
 int roll;
 int cls;
 char *name;
public:
 student(){
 roll=0;
 cls=0;
 name=new char(10);
 strcpy(name,"");
 }
 student(int r, int c, char *n){
 roll=r;
 cls=c;
 name=new char(10);
 strcpy(name,n);
 }
 void enterdata(){

 cout<<"\nEnter Class=";
 cin>>cls;
 }
 void getdata(){
```

```

 cout<<"\nName="<<name<<"\nClass="<<cls<<"\nRoll
="<<roll<<"\n\n";
 }

 ~student(){

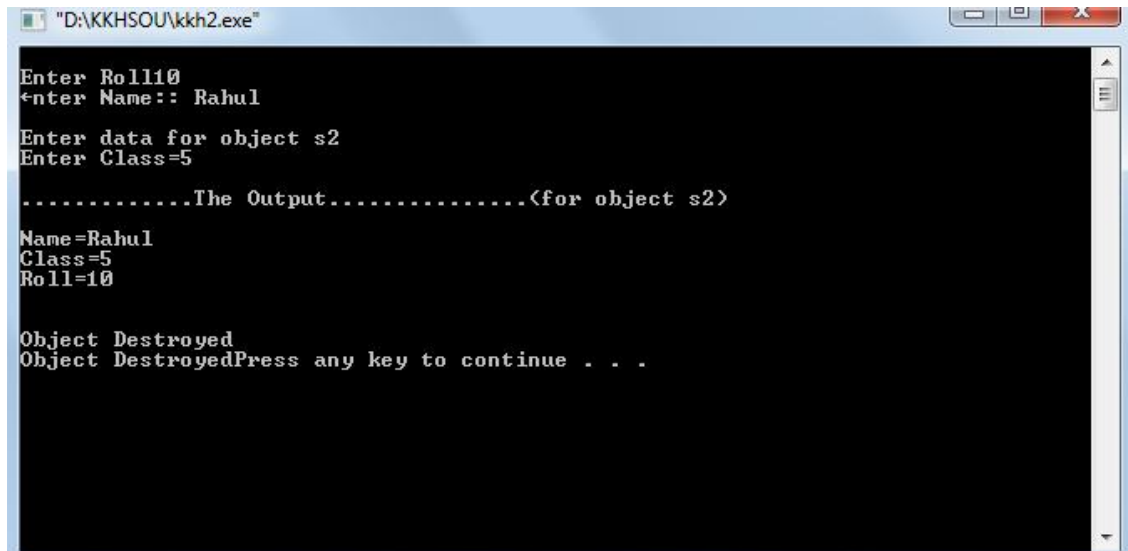
 cout<<"\nObject Destroyed";
 }

};
int main()
{
 student s1;
 int roll;
 char name[20];
 //Enter the intial value of roll and name
 cout<<"\nEnter Roll";
 cin>>roll;
 cout<<"\nEnter Name:: ";
 cin>>name;
 //Dynamic Initialization for Object s2
 student s2(roll,0,name);
 //Enter data for s1 object
 cout<<"\nEnter data for object s2";

 s2.enterdata();
 cout<<"\n.....The Output.....(for object s2)\n";
 s2.getdata();
 return 1;
}

```

The output of the above program



```
"D:\KKHSOU\khh2.exe"

Enter Roll10
Enter Name:: Rahul
Enter data for object s2
Enter Class=5

.....The Output.....<for object s2>

Name=Rahul
Class=5
Roll=10

Object Destroyed
Object DestroyedPress any key to continue . . .
```

If you go through the above program, you will find that the data members of the object s2 is populated using a parameterized constructor. While declaring the object s2 we pass two dynamic values as its parameter (i. e. roll and name), these two values are called dynamic because they are taken as user input at the execution time. This is a very small example, but we can do wonders using this concept. This gives the programmer a very wide range of flexibility.



### CHECK YOUR PROGRESS

- Explain the use of Dynamic initialization of Objects.
- Write a case study for Normal initialization of Object Vs Dynamic initialization of Object

---

## 8.6 SUMMARY

---

In this unit, you have learned that:

- A constructor is a special-member function that initializes the object at the time of its declaration.
- A constructor has the same name as that of the class and is automatically invoked when an object of the class is declared.
- Unlike other member functions, a constructor does not have any return type, not even *void*.
- A constructor can be called implicitly as well as explicitly.
- Constructors are of three types, namely default, parameterized and copy constructor.
- Whenever a class is defined without the constructor definition, the compiler provides a default constructor to construct the object of a class. The default constructor initializes all the data members with garbage values.
- Parameterized constructors can also be used to dynamically initialize the object of a class.
- A copy constructor initialized a new object with values of an existing object of the same class. With copy constructor we can create a clone of an object.
- A destructor is used to free the memory used by an object on exit from the program.

---

## 8.7 KEY TERMS

---

- **Constructor:** A special member function that construct a storage area for the data members of an object by allocating and initializing the memory for them.
- **Destructor:** A special member function that free the memory space used by a object on exit from the program.

- **Parameterized Constructor:** A constructor that accepts one or more parameters at the time of declaration of objects and initializes the data members of the object with these parameters.
- **Copy Constructor:** A constructor that initializes a new object of a class with the values of an existing object of the same class.



## 8.8 MODEL QUESTIONS

---

1. Write down the definition of the following
  - a) Constructor b) Destructor in C++
2. Explain the use of Constructor and Destructor with suitable examples
3. What do you mean by a Copy Constructor? Explain with suitable examples.
4. What is a parameterize constructor? Explain with proper examples.





## 8.9 FURTHER READING

---

- Object Oriented Programming with C++, Third Edition  
-E Balagurusamy
- C++: The Complete Reference, Fourth Edition, Tata McGraw Hill
- *Website for references*



 <http://en.wikipedia.org/wiki/C%2B%2B>

 <http://www.cplusplus.com/doc/tutorial/>

## UNIT-9 : OPERATOR OVERLOADING

### UNIT STRUCTURE

- 9.1 Learning objectives
- 9.2 Introduction
- 9.3 Basic Concept of Overloading
- 9.4 *operator* Keyword
- 9.5 Overloading Unary Operators
- 9.6 Operator Return Type
- 9.7 Overloading Binary Operators
- 9.8 Strings and Operator Overloading
- 9.10 Type Conversion
- 9.11 Let Us Sum Up
- 9.12 Answers to Check Your Progress
- 9.13 Further Readings
- 9.14 Model Questions

---

### 9.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to :

- learn fundamental concept of overloading
- describe the use of the keyword *operator*
- illustrate the overloading of unary and binary operators
- describe manipulation of strings using operators

---

### 9.2 INTRODUCTION

---

So far, we have discussed the concept of class and objects and how to allocate required resource such as memory and initialize the objects of classes using constructors and how to deallocate the memories using destructors. C++ offers another important feature namely *operator over-*

loading, through which operators like +, -, <=, >= etc. can be used with user defined data types, with some additional meaning.

In this unit, we will concentrate on the discussion of overloading of operators (unary and binary) as well as the string manipulations using operators.

---

### 9.3 BASIC CONCEPT OF OVERLOADING

---

We know that operators (+, -, <=, >= etc.) are used to perform operation with the constants and variables. Without the use of the operators a programmer cannot write or build an expression. We have already used these operators with the basic data types such as *int* or *float* etc. The operator +(plus) can be used to perform addition of two variables but we cannot apply the + operator for addition of two objects. If we want to add two objects using the + operator then the compiler will show an error message. To avoid this error message you must have to make aware the compiler about the addition process of two objects. To perform operation with objects you need to redefine the definition of various operators. For example, for addition of objects X and Y, we need to define operator +(plus). Re-defining a operator does not change its original meaning. It can be used for both variables of built-in data types as well as objects of user defined data types.

Operator overloading in C++, permits to provide additional meaning to the operators such as +, \*, >=, -, = etc., when they are applied to user defined data types. Hence, the operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. We will discuss about polymorphism in a next unit. C++ allows the following list of operators for overloading.

Table 9.1 C++ Overloadable Operators

| Operator Category     | Operators                      |
|-----------------------|--------------------------------|
| Arithmetic            | +, -, *, /, %                  |
| Bit-Wise              | &,  , ~, ^                     |
| Logical               | &&,   , !                      |
| Relational            | <, >, ==, !=, <=, >=           |
| Assignment            | =                              |
| Arithmetic assignment | +=, -=, *=, /=, %=, &=,  =, ^= |
| Shift                 | >>, <<, >>=, <<=               |
| Unary                 | ++, --                         |
| Subscripting          | [ ]                            |
| Function call         | ( )                            |
| Dereferencing         | ->                             |
| Unary sign prefix     | +, -                           |
| Allocate and free     | new, delete                    |

---

## 9.4 operator KEYWORD

---

The keyword *operator* helps in overloading of the C++ operators. The general format of operator overloading is shown below :

```

Return Type operator OperatorSymbol ([arg1], [arg2])
{
 // body of the function
}

```

Here, the keyword **operator** indicates that the *OperatorSymbol* is the name of the operator to be overloaded. The operator overloaded in a class is known as *overloaded operator function*.

The following statements shows the use of the *operator* keywords.

```

class Index
{
 // class data and member function
 Index operator ++()
 {
 index temp;
 value = value+1;
 temp.value = value;
 }
}

```

```

 return temp;
 }
};

```

Here, return type of the operator function is the name of a class within which it is declared. It can be defined as follows :

```

class Index
{
 // class data and member function
 Index operator ++();
};

```

Index Index :: operator ++( )

```

{
 index temp;
 value = value+1;
 temp.value = value;
 return temp;
}

```

The operator function should be either a member function or a friend function. When the operator function is declared as member function and takes no argument, it is known as *unary operator overloading* and when it takes one argument it is known as *binary operator overloading*.

---

## 9.5 OVERLOADING UNARY OPERATORS

---

When an operator function takes no argument, it is called as unary operator overloading. You are already familiar with the operators ++, --, and -, which have only single operands are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions. The following program demonstrates the overloading of unary '--' operators.

### // Program 9.1

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

```

#include<conio.h>

class unary
{
 int x, y, z;

 public :
 unary (int i, int j, int k) // parameterized constructor
 {
 x=i; y=j; z=k;
 }
 void display(); //displays contents of member variables
 void operator --(); //overloads the unary operator --
};

void unary :: operator --()
{
 --x; --y; --Z; // values of variables will be decremented by 1
}

void unary :: display()
{
 cout<<"X="<<x<<"\n";
 cout<<"Y="<<y<<"\n";
 cout<<"Z="<<z<<"\n";
}

void main()
{
 clrscr();
 unary A(31, 41, 51);
 cout<<"\n Before Decrement of A :\n";
 A.display();
 --A; // calls the function operator --()
 cout<<"\n After Decrement of A :\n";
 A.display();
 getch();
}

```

**RUN :** Before Decrement of A : X=31

Y=41

Z=51

After Decrement of A : X=30

Y=40

Z=50

---

## 9.6 OPERATOR RETURN TYPE

---

In the above example, we have declared the operator function of type *void* i.e. it will not return any value. But it is possible to return value and assign it to other object of same type. The return value of the operator is always of the class type, it means that class name will be in the place of the return type specification because we are applying the operator overloading properties only for the objects. Always remember that an operator cannot be overloaded for basic data types, so the return value of operator function will be of class type. The following program demonstrates the operator return types :

### // Program 9.2

```
#include<iostream.h>
#include<conio.h>
class unary
{
 int x;
public :
 unary () { x=0; }
 int getx() // returns the current value of variable x
 { return x; }
 unary operator ++();
};
unary unary :: operator ++()
{
 unary temp;
 x=x+1;
 temp.x=x;
 return temp;
}
```

```
 }
 void main()
 {
 clrscr();
 unary A1,A2;
 cout<<"\n A1="<<A1.getx();
 cout<<"\n A2="<<A2.getx();
 A1=A2++; // first increment the value of A2 and assigns it to A1
 cout<<"\n A1="<<A1.getx();
 cout<<"\n A2="<<A2.getx();
 A1++; // object A1 is increased
 cout<<"\n A1="<<A1.getx();
 cout<<"\n A2="<<A2.getx();
 getch();
 }
```

**RUN :** A1 = 0

A2 = 0

A1 = 1

A2 = 1

A1 = 2

A2 = 1

---

## 9.7 OVERLOADING BINARY OPERATORS

---

Binary operators are overloaded by using member functions and friend functions. The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one. When the function defined for the binary operator overloading is a friend function, then it uses two arguments. Here, we will discuss the overloading of binary operator when the operator function is a member function.

Binary operator overloading, as in unary operator overloading, is performed using a keyword *operator*. The following program demonstrates the



overloading of binary operators.

**// Program 9.3**

```
#include <iostream.h>
#include<conio.h>

class Binary
{
 private:
 int x;
 int y;
 public:
 Binary() //Constructor
 { x=0; y=0; }
 void getvalue() //Member Function for Inputting Values
 {
 cout <<"\n Enter value for x:";
 cin >> x;
 cout << "\n Enter value for y:";
 cin>> y;
 }
 void displayvalue() //Member Function for Outputting Values
 {
 cout<<"\n\nThe resultant value : \n";
 cout <<" x =" << x <<" ; y ="<<y;
 }
 Binary operator +(Binary);
};

Binary Binary :: operator +(Binary e2)
//Binary operator overloading for + operator defined
{
 Binary temp;
 temp.x = x+ e2.x;
 temp.y = y+e2.y;
 return (temp);
}

void main()
{
```

```

Binary e1,e2,e3; //Objects e1, e2, e3 created
clrscr();
cout<<"\nEnter value for Object e1:";
e1.getvalue();
cout<<"\nEnter value for Object e2:";
e2.getvalue();
e3= e1+ e2; //Binary Overloaded operator used
e3.displayvalue();
getch();
}

```

**RUN :** Enter value for Object e1 :

```

Enter value for x : 10
Enter value for y : 20
Enter value for Object e2 :
Enter value for x : 30
Enter value for y : 40
The Resultant Value : x = 40; y = 60

```

In the above example, the class Binary has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Binary. The definition is performed outside the class Binary by using the scope resolution operator and the keyword *operator*.

The important aspect is the statement:

```
e3= e1 + e2; // e3= e1.operator +(e2)
```

The binary overloaded operator '+' is used. When the compiler encounters such expressions, it examines the argument type of the operator. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Binary in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' i.e. the expression means **e3 = e1.operator +(e2)**. The operator returns a value (binary object temp in this case), which can be assigned to another object (e3 in this case).

Since the object `e2` is passed as argument to the operator '+' inside the function defined for binary operator overloading, the values are accessed as `e2.x` and `e2.y`. This is added with `e1.x` and `e1.y`, which are accessed directly as `x` and `y`.

Always remember that, in the overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument to the operator function.

---

## 9.8 STRINGS AND OPERATOR OVERLOADING

---

We are already familiar with the `strcat()` function which is used for concatenation of strings. Consider the following two strings

```
char str1[50] = "Bachelor of Computer";
char str2[20] = "Application";
```

The string `str1` and `str2` are combined, and the result is stored in `str1` by invoking the function `strcat()` as follows :

```
strcat(str1, str2);
```

The same operation can be done by defining a string class and overloading the `+` operator. The following program demonstrates the concatenation of two string using the overloading concept.

### // Program 9.4

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class String
{
 private:
 char str[100];
 public:
```

```

 String() //Constructor
 { strcpy(str, " "); }

String(char *msg) //Constructor
{ strcpy(str, msg); }

void display() //Member Function for Display strings
{
 cout <<str;
}

String operator +(String s);
};

String String :: operator +(String s)
//Binary operator overloading for + operator defined
{
 String temp = str;
 strcat(temp.str, s.str);
 return temp;
}

void main()
{
 clrscr();
 String str1 = "Bachelor of Computer";
 String str2 = "Application";
 String str3;
 str3= str1+str2;
 cout<<"\n str1 =";
 str1.display();
 cout<<"\n str2 =";
 str2.display();
 cout<<"\n The String after str3=str1+str2 \n \n";
 str3.display();
 getch();
}

```

In this program, the concatenation is performed by creating a temporary string object *temp* and initializing it with the first string. The second string is

added to first string in the object temp using the `strcat()` and finally the resultant temporary string object temp is returned. Here, in the program, the length of `str1+str2` should not exceed the array size 100 (i.e. `char str[100]`).

Thus, we have seen that, in C++ programming language, operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary. The operators that cannot be overloaded are - `., ?:, sizeof, ::, .*, #, ##`.

---

## 9.10 TYPE CONVERSION

---

We cannot convert between user-defined data types(classes) just like we convert between basic types. This is because the compiler does not know anything about the user-defined type.

Now, let us look into how C++ handles conversions for its **built-in types (int, float, char, double etc.)**. When you make a statement assigning a value of one standard type to a variable of another standard type, C++ automatically will convert the value to the same type as the receiving variable, providing the two types are compatible.

For example, the following statements all generate numeric type conversions:

```
long count = 8; // int value 8 converted to type long
double time = 11; // int value 11 converted to type double
int side = 3.33; // double value 3.33 converted to type int 3
```

These assignments work because C++ recognizes that the diverse numeric types all represent the same basic thing, a number, and because C++ incorporates built-in rules for making the conversions. However, you can lose some precision in these conversions. For example, assigning 3.33 to

the int variable that side results in that side getting the value 3, losing the 0.33 part.

The C++ language does not automatically convert types that are not compatible.

For example, the statement

```
int * p = 10; // type clash
```

fails because the left-hand side is a pointer-type, whereas the right-hand side is a number. And even though a computer may represent an address internally with an integer, integers and pointers conceptually are quite different. For example, you wouldn't square a pointer. However, when automatic conversions fail, you may use a type cast:

```
int * p = (int *) 10; // ok, p and (int *) 10 both pointers
```

This sets a pointer to the address 10 by type casting 10 to type pointer-to-int (that is, type int \*).

Now, let us look into how C++ handles conversions from basic type to user-defined types vice-versa.

### **Basic type to user defined type :**

This type of conversion can be easily carried out. It is automatically done by the compiler with the help of in-built routines or by type casting. In this type the left hand operand of = sign is always class type or user defined type and the right hand side operand is always basic type. The following program explains this type of conversion.

#### **//Program 9.5**

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

```
#include<conio.h>
```

```
class Test
```

```
{
```

```
 private:
```

```
 int x;
```

```
 float y;

 public:
 Test() //Constructor
 { x=0; y=0; }
 Test(float z) //Constructor with one argument
 { x=2; y=z; }
 void display() // Function for displaying values
 {
 cout <<"\n x =" << x <<" y ="<<y;
 cout <<"\n x =" << x <<" y ="<<y;
 }
};

void main()
{
 Test a;
 clrscr();
 a=9;
 a.display();
 a=9.5;
 a.display();
 getch();
}
```

**RUN :** x=2 y=9  
x=2 y=9  
x=2 y=9.5  
x=2 y=9.5

In the above program, the class Test has two member variable of type integer and float. It also has two constructors one with no arguments and the second with one arguments. In *main()* function, **a** is an object of class Test. When a is created the constructor with no argument is called and data members are initialize to zero. When a is initialize to 9 the constructor with float argument i.e. **Test(float z)** is invoked. The integer value is converted to float type and assigned member variable y. Again when a is assigned to 9.5, same process repeated. Thus, the conversion from basic

to class type is carried out.

### User defined type to basic type :

As we know, the compiler does not have any prior information about user defined data type using class, so in this type of conversion it needs to inform the compiler how to perform conversion from class to basic type. For this purpose, a conversion function should be defined in the class in the form of the operator function. The operator function is defined as an overloaded basic data type which takes no arguments. The syntax of such a conversion function is shown below-

```
operator Basic type()
{
 // steps for converting
}
```

In the above syntax, you have noticed that, the conversion function has no return type. While declaring the operator function the following condition should always remember :

- i) the operator function should not have any argument.
- ii) it has no any return type.
- iii) it should be a class member.

The following program demonstrates this conversion mechanism :

#### // Program 9.6

```
#include <iostream.h>
#include<conio.h>

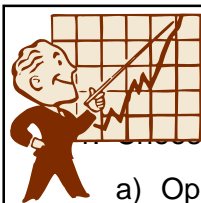
class Time
{
 private:
 int hour;
 int minute;
 public:
 Time(int a)
```



```
 { hour=a/60;
 minute=a%60;
 }
operator int()
{
 int a;
 a=hour*60+minute;
 return a;
}
};
void main()
{
 clrscr();
 Time t1(500);
 int i=t1; // operator int() is invoked
 cout<<"\n"<<"The value of i:"<<i;
 getch();
}
```

**RUN :** The value of i : 500

In the above program, the statement ***int i=t1***, invokes the operator function which finally converts a time object to corresponding magnitude (of type int).



### CHECK YOUR PROGRESS

Choose the correct answer from the following :

- a) Operator overloading is
  - i) making C++ operators work with objects
  - ii) giving C++ operators more than they can handle
  - iii) giving new meaning to existing C++ operators
  - iv) making new C++ operators
- b) To convert from a user-defined class to basic type, you would use

- i) a built-in conversion function
  - ii) a one argument constructor
  - iii) an overloaded = operator
  - iv) a conversion function that is a member of the class
- c) To convert from a basic type to user-defined class, you would use
- i) a built-in conversion function
  - ii) a one argument constructor
  - iii) an overloaded = operator
  - iv) a conversion function that is a member of the class
- d) \_\_\_\_\_ operator must have one class object.
- i) +
  - ii) new
  - iii) all
  - iv) none of these
- e) Binary overload operators are passed \_\_\_\_\_ arguments.
- i) one
  - ii) two
  - iii) no
  - iv) none of the above

2. Fill in the blanks :

- i) The statement  $x=y$  will cause \_\_\_\_\_ if the objects are of different classes.
- ii) \_\_\_\_\_ is making operators to work with user defined data types.
- iii) Single argument constructor is usually defined in the \_\_\_\_\_ class.
- iv) \_\_\_\_\_ function must not have a return type.
- v) \_\_\_\_\_ are operators that act on only one operand.

## 9.11 LET US SUM UP

1. Operator overloading is one of the important concepts in C++ which allows to provide additional meaning to operators +, -, >=, <= etc. when they are applied to user defined data types.
2. Overloaded operators are redefined within a class using the keyword **operator** followed by an operator symbol. When an operator is

overloaded, the produced symbol is called the operator function name.

3. Overloading of operator cannot change the basic meaning of an operator. When an operator is overloaded, its properties like syntax, precedence and associativity remain constant.
4. Operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions.
5. The binary operators require two operand. Binary operators are overloaded by using member functions and friend functions.
6. The operators which cannot be overloaded are - ., ?:, sizeof, ::, \* , #, ##.
7. The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types.
8. Actually there are three possibilities of data conversion :
  - a) Basic type to user defined type(class type)
  - b) User defined type(class type) to basic type
  - c) Class type to another class type (we have not discussed here)



## 9.12 ANSWERS TO CHECK YOUR PROGRESS

1. a) iii,                      b) iv,                      c) iii,  
d) ii,                      e) i.
2. i) Compiler error,  
ii) operator overloading,  
iii) destination,  
iv) casting operator,  
v) unary operator



## 9.13 FURTHER READINGS

1. Object Oriented Programming with C++, E. Balagurusamy, Tata McGraw Hill Publication.
2. The Complete Reference C++, Herbert Schildt, Tata McGraw Hill Publication.

3. Mastering C++, K.R. Venugopal, Rajkumar, T. Ravi Shankar, Tata McGraw Hill Publication.



---

## 9.14 MODLE QUESTIONS

---

1. What is operator overloading ? Give the advantage of operator overloading.
2. What is operator function ? Describe operator function with syntax and examples.
3. What is the difference between overloading of binary operators and unary operators ?
4. Explain the conversion from basic type to user defined type(class type) with examples.
5. Explain the conversion from user defined type(class type) to basic type with examples.
6. Write a program to overload the -- operator.
7. Write a program to overload the binary operator + in order to perform addition of complex numbers.
8. Write a program to overload the relational operator (>, <, ==) in order to perform the comparision of two strings.

## UNIT-10 : INHERITANCE

### UNIT STRUCTURE

- 10.1 Learning Objectives
- 10.2 Introduction
- 10.3 Inheritance
  - 10.3.1 Defining a Derived Class
  - 10.3.2 Accessing Base Class Members
- 10.4 Types of Inheritance
  - 10.4.1 Single Inheritance
  - 10.4.2 Multiple Inheritance
  - 10.4.3 Hierarchical Inheritance
  - 10.4.4 Multilevel Inheritance
  - 10.4.5 Hybrid Inheritance
  - 10.4.6 Multipath Inheritance
- 10.5 Virtual Base Classes
- 10.6 Abstract Classes
- 10.7 Let Us Sum Up
- 10.8 Answers to Check Your Progress
- 10.9 Further Readings
- 10.10 Possible Questions

---

### 10.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to :

- learn about the concept of inheritance in C++
- create new classes by reusing the members and properties of existing classes
- learn the advantages and disadvantages of inheritance in programming
- learn how to use base class access specifier *public*, *private* and *protected*
- learn the use of virtual base class and abstract class

---

## 10.2 INTRODUCTION

---

In this unit, we shall discuss one important and useful feature of Object - Oriented Programming (OOP) which is called **inheritance**. C++ supports the concept of inheritance. We have already discussed the concept of classes and objects in our previous unit which are prerequisite for this unit.

With the help of inheritance we can reuse (or inherit) the property of a previously written class in a new class. There are different types of inheritance which will be discussed in this unit. The concept of abstract and virtual base class will also be covered in this unit.

---

## 10.3 INHERITANCE

---

In Biology, *inheritance* is a term which represents the transformation of the hereditary characters from parents or ancestors to their descendent. In the context of Object-Oriented Programming, the meaning is almost same. The process of creating a new class from existing classes is called **inheritance**. The newly created class is called **derived class** and the existing class is called the **base class**. The derived class inherits some or all of the characteristics of the base class. Derived class may also possess some other characteristics which are not in the base class.

For example, let us consider two classes namely, “employee ” and “manager”. Whatever information is present in “employee” class, the same will be present in “manager” also. Apart from that there will be some extra information in “manager” class due to other responsibility assigned to managers. Due to the facility of inheritance in C++, it is enough only to indicate those pieces of information which are specific to manager in its class. In addition, the “manager” class will inherit the information of “employee” class.

Before discussing the different types of inheritance and their implementation in C++, we will first denote the advantages and disadvantages of inheritance.

### **Advantages of Inheritance**

- **Reusability:**

Reusability is an important feature of Object Oriented Programming. We can reuse code in many situations with the help of inheritance. The base class is defined and once it is compiled, it need not be rewritten. Using the concept of inheritance the programmer can create as many derived classes from the base class as needed. New features can also be added to each derived class when required.

- **Reliability and Cost:**

Reusability would not only save time but also increase reliability and decrease maintenance cost.

- **Saves Time and Effort:**

The reuse of class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

### **Disadvantages of Inheritance**

- Inappropriate use of inheritance makes a program more complicated.
- In the class hierarchy various data elements remain unused, the memory allocated to them is not utilized.

---

## **10.3.1 DEFINING A DERIVED CLASS**

---

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of deriving a new class from an existing class is as follows:

```
class DerivedClassName : access specifier BaseClassName
{

 //members of derived class

};
```

The derived class name and the base class name is separated by a colon “:”. We can derive classes using any of the three base class access specifiers: **public**, **private** or **protected**. If we do not specify the access specifier, then by default it will be **private**. Generally, it is convenient to specify the access specifier while deriving a class. Access specifiers are sometime called *visibility mode*. It determines the access control of the base class members inside the derived class. In the previous unit, we have already learnt **private** and **public** access specifier while declaring classes. **Protected** specifier has a significant role in inheritance.

---

### 10.3.2 ACCESSING BASE CLASS MEMBERS

---

There may be three types of base class derivation:

- Public derivation
- Private derivation
- Protected derivation

- **Public derivation**

When the base class is inherited by using **public access specifier** then all public members of the base class become public members of the derived class, and all protected members of the base class become protected members of the derived class. The private members of the base class remain private and are not accessible by members of derived class. Let us examine this with the following example:

// **Program 10.1:** Base class with public access specifier

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

```
#include<conio.h>
```

```
class base
```

```
{
```

```
 private:
```

```
 int num1,num2;
```



```
 public:
 void input(int n1, int n2)
 {
 num1=n1;
 num2=n2;
 }

 void display()
 {
 cout<<"Number 1 is: "<<num1<<endl;
 cout<<"Number 2 is: "<<num2;
 }
 }; //end of base class

class derived : public base
{
 private:
 int num3;
 public:
 void enter(int n3)
 {
 num3=n3;
 }
 void show()
 {
 cout<<"\nNumber 3 is: "<<num3;
 }
}; //end of derived class

int main()
{
 derived d; // d is an object of derived class
 clrscr();
 d.enter(15); //enter() function is called by object d
 d.input(5,10); /* accessing base class member. input() is a public
 member of base class */
 d.display(); /* accessing base class member display() is a public
 member of base class */
}
```

```
d.show();
getch();
return 0;
}
```

The output will be like this:

```
Number 1 is : 5
Number 2 is : 10
Number 3 is : 15
```

Here, **d** is a derived class object. With the statement **d.input(5,15);** we have made a call to the function *input()* of base class by the derived class object. Similarly, we have called *display()* member function of base class.

- **Private derivation**

When the base class is inherited by using **private access specifier**, all the public and protected members of the base class become private member of the derived class. Therefore, public members of base class can only be accessed by the member functions of the derived class and they are not accessible to the objects of the derived class.

For example, if we use the statement

**class derived : private base**

instead of **class derived : public base**

in the above program, it will give two error message while compiling:

Error: base::input(int,int) is not accessible

Error: base::display() is not accessible

As the base class is privately inherited, *input()* and *display()* become private to the derived class although they were public in the base class. So other functions like *main()* cannot access them. Statement like **d.input(5,10);** and **d.display();** will be invalid in that case.

**Protected Members and Inheritance:**

Protected members provide greater flexibility in case of inheritance. By using **protected** instead of private declaration, we can create class members that are private to their class but that can still be inherited and accessed by derived class. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It also becomes ready for further inheritance. If a base class is inherited as **private**, then the **protected** member of base class becomes **private** in the derived class. Although it is available to the member function of the derived class, it is not available for further inheritance since **private** members cannot be inherited.

**/\*Program 10.2:** Program showing protected members inherited in public mode \*/

```
#include<iostream.h>
#include<conio.h>
class base
{
 protected:
 int num1,num2; /*private to base, protected to derived and
 accessible by derived class member function*/
 public:
 void input(int n1, int n2)
 {
 num1=n1;
 num2=n2;
 }
 void display()
 {
 cout<<"Number 1 is: "<<num1<<endl;
```

```

 cout<<"Number 2 is: "<<num2;
 }
}; //end of base class

class derived : public base //base class is publicly inherited
{
 private:
 int s;
 public:
 void add()
 {
 s=num1+num2 ; /* derived class accessing base class
 protected member num1, num2 */
 }
 void show()
 {
 cout<<"\nSummation is : "<<s;
 }
}; //end of derived class

int main()
{
 derived d; // d is an object of derived class
 clrscr();

 d.input(10,20); /* accessing base class member. input() is a function
 of base class */

 d.display(); /* accessing base class member. display() is a function of
 base class */

 d.add();
 d.show();
 getch();
 return 0;
}

```

The above program will give the output as:

```

Number 1 is : 10
Number 2 is : 20
Summation is : 30

```

The derived class member function *void add()* can access *num1* and *num2* of the base class as because *num1* and *num2* are declared as *protected* and base class access specifier is *public*.

- **Protected derivation**

When the base class is inherited by using ***protected access specifier***, then all protected and public members of base class become protected members of the derived class. Let us consider the following example:

**//Program 10.3** : Base class derived as protected

```
#include<iostream.h>
#include<conio.h>
class base
{
 protected:
 int num1,num2;
 public:
 void input(int n1, int n2)
 {
 num1=n1;
 num2=n2;
 }
 void display()
 {
 cout<<"Number 1 is: "<<num1<<endl;
 cout<<"Number 2 is: "<<num2;
 }
}; //end of base class

class derived : protected base
{
 private:
 int s;
 public:
 void add()
 {
```

```

 input(30,60); /*member function add() of derived
 class can access input() as it is inher-
 ited as protected */

 s=num1+num2; // num1,num2 are inherited as
 // protected, so add() can access
 }
 void showall()
 {
 display(); //display() is inherited as protected
 cout<<"\nSummation is : "<<s;
 }
}; //end of derived class

int main()
{
 derived d;
 clrscr();
 //d.input(10,20); /* invalid. input() is inherited as protected member of
 derived. main() can't access it */

 d.add(); /* accessing base class member display() is a function of base
 class */

 d.showall();// public member of derived
 // d.display(); /* invalid. display() can be accessible by derive class
 member function only */

 getch();
 return 0;
}

```

The above program will give the following output:

```

Number 1 is : 30
Number 2 is : 60
Summation is : 90

```

In the program we can see that *input()*, *display()*, *num1*, *num2* of base class are inherited as protected to derive class. The member functions *add()*, *showall()* of derived class can use them; as protected members are accessible by derive class members. But *main()* is not a member function and it cannot access *input()* and *display()*. Although *num1*, *num2* are protected to derived class, but they behave as private to base class.



## CHECK YOUR PROGRESS -1

1. Answer the following by selecting the appropriate option:
  - (i) By using protected, one can create class members that
    - (a) cannot be inherited and accessed by a derived class
    - (b) can be accessed by a derived class
    - (c) can be public
    - (d) none of these
  - (ii) Class members are by default \_\_\_\_\_
    - (a) protected
    - (b) public
    - (c) private
    - (d) none of these
  - (iii) When base class access specifier is protected, then public members of base class can be accessible by
    - (a) member function of derived class
    - (b) main() function
    - (c) objects of derived class
    - (d) none of these
  - (iv) Private data members of base class can be inherited by declaring them as
    - (a) private
    - (b) public
    - (c) protected
    - (d) none of these
  - (v) If we donot specify the visibility mode in base class derivation then by default it will be
    - (a) protected
    - (b) private
    - (c) public
    - (d) none of these
  - (vi) Private data members can be accessed
    - (a) from derive class
    - (b) only from the base class itself
    - (c) both from the base class and from its derived class
    - (d) None of these

---

## 10.4 TYPES OF INHERITANCE

---

A program can use one or more base classes to derive a single class. It is also possible that one derived class can be used as base class for another class. Depending on the number of base classes and levels of derivation inheritance is classified into the following forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

---

### 10.4.1 SINGLE INHERITANCE

---

The programs discussed so far in this unit are examples of single inheritance. In **single inheritance** the derived class has only one base class and the derived class is not used as base class. The pictorial representation of single inheritance is given below.

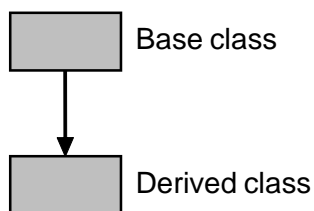


Fig. 10.1: Single Inheritance

The arrow directed from base class towards the derived class indicates that the features of base class are inherited to the derived class. In the following program, we have derived "**employee**" class from "**person**" class. Data member "**name**" and "**age**" are common to both of the two classes. "**name**" and "**age**" are declared as **protected** so that derive class can inherit "**name**" and "**age**" from the base class "**person**". Base class is inherited in public mode. Employee



may have some other data like designation, salary. So the other data members of “**employee**” class are “**desig**” and “**salary**”.

**/\* Program 10.4:** Single inheritance with protected data member and public inheritance of base class \*/

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class person //base class "person"
{
 protected:
 char name[30]; //protected to derived class 'employee'
 float age;
 public:
 void enter(char *nm, float a) /* base class member function */
 {
 strcpy(name,nm);
 age=a;
 }

 void display() //base class member function
 {
 cout<<"Name: "<<name<<endl;
 cout<<"Age: "<<age<<endl;
 }
};

class employee : public person /*base class "person" is publicly
 inherited */
{
 /*by derived class "employee" */
 private:
 float salary;
 char desig[20];

 public:
 void enter_data(char *n,char *d,float ag,float s)
 {
```

```

 strcpy(name,n); // "name" of base class can be accessi-
 ble
 //by derived class member function

 strcpy(desig,d);
 salary=s;
 age=ag; //age can be accessible by
 //enter_data() of derived class
 }

 void display_all() //derived class member function
 {
 display(); //can be used here as publicly inherited
 cout<<"Designation: "<<desig<<endl;
 cout<<"Salary: "<<salary<<endl;
 }
};

int main()
{
 employee e1,e2; //e1,e2 are objects of derived class "employee"
 person p; // p is an object of base class "Person"
 clrscr();

 e1.enter_data("Raktim","Clerk",32,5000);
 cout<<"Employee Details....."<<endl;
 e1.display_all();

 e2.enter("Vaskar",41); /*erived class object e1 accessing public
 member of base enter() */

 e2.display(); /*derived class object e2 accessing public member of
 base display() */

 cout<<endl<<"Person Details....."<<endl;
 p.enter("Pragyan",24);
 p.display();
 getch();
 return 0;
}

```

Here, the derived class “**employee**” uses **name** and **age** of base class “**person**” with the help of derived class member function **enter\_data()**.

Two different classes may have member functions with the same name as well as same set of arguments. But in case of inheritance, an ambiguous situation arises when base class and derived classes contain member functions with same name. In `main()`, if we call member function of that particular name of base class with derived class object, then it will always make a call to the derived class member function of that name. This is because, the function in the derived class *overrides* the inherited function. However, we can invoke the function defined in base class by using the **scope resolution operator (::)** to specify the class. For example, let us consider the following program.

**/\*Program 10.5:** when base and derived class has member functions with same name\*/

```
#include<iostream.h>
#include<conio.h>
class B
{
 protected:
 int p;
 public:
 void enter()
 {
 cout<<"\nEnter an integer:";
 cin>>p;
 }
 void show()
 {
 cout<<"\n\nThe number in Base Class is: "<<p;
 }
};
class D : public B
```

```

{
 private:
 int q,r;
 public:
 void enter()//overrides enter() of "B"
 {
 B::enter();
 cout<<"\nEnter an integer:";
 cin>>q;
 }
 void show()
 {
 r=p*q;
 cout<<"\nEnter numbers in Base and
 Derived class are:"<<p<<"\t"<<q;
 cout<<"\n\nThe product is : "<<r;
 }
};

int main()
{
 D d; //d is an object of class derived class "D"
 clrscr();
 d.enter(); //invokes enter() of "D"
 d.show(); //invokes show() of "D"
 d.B::show(); //invokes show() of "B"
 getch();
 return 0;
}

```

In the program, the function name *show()* is same in both base “B” and derived class “D”. To call *show()* of base class “B”, we have used the statement **d.B::show()**; If we use simply **d.show()**; then it will invoke *show()* of derived class “D”.

When a derive class implements a function that has the same name as well as the same set of arguments as the function in the base

class, it is called **function overriding**. When such a function is called through a object of derived class, then the derived class function would be invoked. However, that function in base class would remain hidden.

But there are certain situations where function overriding plays an important role.

---

### 10.4.2 MULTIPLE INHERITANCE

---

When one class is derived from two or more base classes then it is called **multiple inheritance**. This type of inheritance allows us to combine the properties of more than one existing classes in a new

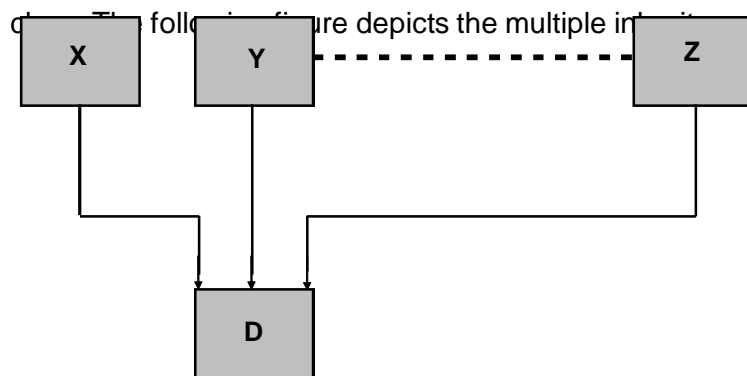


Fig.10.2: Multiple inheritance

We have to specify the base classes one by one separated by commas with their access specifiers. The general form of deriving a derived class from numbers of base class is as follows:

```
class D : public X, public Y, public Z
{
 //body of the derived class
};
```

where X,Y, Z are base classes and D is the derived class. There may be numbers of base classes in multiple inheritance which is indicated

by the dotted line in the figure.

For demonstration of multiple inheritance let us consider the following program. There are three base classes and one derived class. The derived class CHARACTER has one private member “n” and two public member functions “enter()” and “show()”. The function “enter()” is used to read a number, a vowel, a consonant and a symbol from the keyboard and the “show()” function is used to display the contents on the screen. The class members of all the three base classes are publicly derived.

// **Program 10.6** : Example of Multiple inheritance

```
#include<iostream.h>
#include<conio.h>
class V //base class
{
 protected:
 char v;
};
class C //base class
{
 protected:
 char c;
};
class S //base class
{
 protected:
 char s;
};

class CHARACTER : public V, public C, public S
{
 private:
 int n;
 public:
 void enter() //derived class member function
 {
 cout<<"\nEnter a vowel:";
 cin>>v; //accessing protected member v of class
```

```

 "VOWEL "
 cout<<"\nEnter a consonent:";
 cin>>c; //accessing c of "CONSONENT" class
 cout<<"\nEnter a symbol:";
 cin>>s //accessing s of "SYMBOL" class
 cout<<"\nEnter a number:";
 cin>>n; //accessing n of "NUMBER" class
 }
 void show()
 { cout<<"\nThe entered characters are :\n\n";
 cout<<"\nVowel: "<<v;
 cout<<"\nConsonent: "<<c;
 cout<<"\nSymbol: "<<s;
 cout<<"\nNumber: "<<n;
 }
};
int main()
{
 CHARACTER o; //o is an object of derived class "character"
 clrscr();
 o.enter();
 o.show();
 getch();
 return 0;
}

```

One switable example of the implementation of multiple inheritance is shown in the program below:

```

/*Program 10.7: Program showing multiple inheritance with two base
class (practical, theory) and one derived class (result) */
#include<iostream.h>
#include<conio.h>
class practical //base class "practical"
{
 protected:

```

```

 float p1_marks, p2_marks, total;
 public:
 void practical_marks()
 {
 cout<<"Enter marks of practical paper 1 and paper
 2: ";
 cin>>p1_marks>> p2_marks;
 }
 float add() //returns the total of practical
 {
 total=p1_marks+p2_marks;
 return total;
 }
 void display_practical()
 {
 cout<<endl<<"Total Practical marks:"<<total;
 }
}; //end of "practical" class

class theory // base class "theory"
{
 protected:
 float phy, chem, math, total_marks;
 public:
 void theory_marks(){
 cout<<"Enter marks of Physics, Chemistry and
 Mathematics:";
 cin>>phy>>chem>>math;
 }
 float sum()
 {
 total_marks=phy+chem+math;
 return total_marks;
 }
 void display_theory() {
 cout<<endl<<"Total Theory marks:"<<total_marks;
 }
}; //end of base class "theory"

```



```
class result : public practical,public theory
{
 protected:
 int roll;
 float grand_total,t,p;
 public:
 void enter() {
 cout<<"ENTER STUDENT
 INFORMATION....."<<endl;
 cout<<"Enter Roll no.: ";
 cin>>roll;
 theory_marks(); //inherited publicly from base class
 "theory"
 practical_marks(); //inherited from base class "practical"
 }
 void theory_practical()
 {
 t=sum();
 p=add();
 grand_total=t+p;
 cout<<"\nThe total marks of the student is:"
 <<grand_total;
 }
}; //end of derived class "result"

int main()
{
 result s1; //object of derived class
 clrscr();
 s1.enter();
 s1.theory_practical(); // accessing derived class member
 s1.display_theory(); //s1 accessing "display_theory()" of "theory"
 s1.display_practical();
 getch();
 return 0;
}
```

```
}
```

When we execute the program entering marks for practical and theory papers for a particular student, then it will display the result as follows:

ENTER STUDENT INFORMATION.....

Enter Roll no.: 1

Enter marks of Physics, Chemistry and Mathematics: 65 72 81

Enter marks of practical paper 1 and paper 2 : 25 26

The total marks of the student is : 269

Total Theory marks : 218

Total Practical marks : 51

The above program consists of three classes: two base classes ("**practical**" and "**theory**") and one derived class ("**result**"). The member function "**enter()**" of derive class inherits member functions "**practical\_mark()**" and "**theory\_marks()**" of base class "**practical**" and "**theory**" respectively. Similarly, member function "**theory\_practical()**" uses "**sum()**" and "**add()**" of base class to calculate the "**grand\_total**" marks of student. Thus, in the derived class we need not have to write functions for entering practical and theory marks. We just inherit them from the base classes.



### EXERCISE-1

Q. Suppose a class D is derived from class B. B has two public member functions *getdata()* and *showdata()* and D has two public functions *readdata()* and *displayall()*. Define the classes such that both function *getdata()* and *showdata()* should be accessible in the *main()* function.

## 10.4.3 HIERARCHICAL INHERITANCE

Derivation of more than one classes from a single base class is termed

as **hierachical inheritance**. This is a very common form of inheritance in practice. The rules for defining such classes are the same as in single inheritance. The pictorial representation of hierarchical inheritance is as follows:

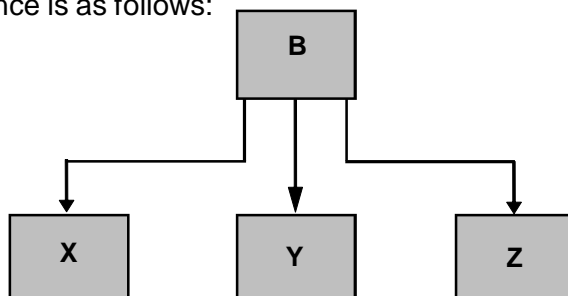


Fig.10 3: Hierarchical Inheritance

For demonstration of hierarchical inheritance let us consider a program with one base class ("student") and three derived classes ("arts", "science" and "commerce").

**//Program 10.8:** Demonstration of hierarchical inheritance

```

#include<iostream.h>
#include<conio.h>
class student // base class "student"
{
 protected:
 char fname[20],lname[20];
 int age,roll;
 public:
 void student_info()
 {
 cout<<"Enter the first name and last name: ";
 cin>>fname>>lname;
 cout<<"\nEnter the Roll no.and Age: ";
 cin>>roll>>age ;
 }
 void display()
 {
 cout<<"\nRoll Number = "<<roll;
 cout<<"\nFirst Name = "<<fname<<"\t"<<lname;
 cout <<"\nAge = " << age;
 }
 };

```

```
 }
};
class arts : public student //derived class arts
{
 private:
 char asub1[20], asub2[20], asub3[20] ;
 public:
 void enter_arts()
 {
 student_info(); //base class member function
 cout << "\n Enter the subject1 of the arts student:";
 cin >> asub1 ;
 cout << "\n Enter the subject2 of the arts student:";
 cin >> asub2 ;
 cout << "\n Enter the subject3 of the arts student:";
 cin >> asub3 ;
 }
 void display_arts()
 {
 display();//base class member function
 cout<< "\n\t Subject1 of the arts student="<< asub1;
 cout<< "\n\t Subject2 of the arts student="<< asub2;
 cout<< "\n\t Subject3 of the arts student="<< asub3;
 }
};
class commerce : public student //derived class "commerce"
{
 private:
 char csub1[20], csub2[20], csub3[20] ;
 public:
 void enter_com(void)
 {
 student_info(); //base class member function
 cout << "\t Enter the subject1 of the commerce
student: ";
 cin >> csub1;
 cout << "\t Enter the subject2 of the commerce
```

```
 student: ";
 cin >> csub2 ;
 cout << "\t Enter the subject3 of the commerce stu-
 dent: ";
 cin >> csub3 ;
 }
 void display_com()
 {
 display(); //base class member function
 cout<<"\nSubject1 of the commerce student="
 <<csub1;
 cout <<"\nSubject2 of the commerce student = "
 <<csub2;
 cout<<"\nSubject3 of the commerce student = " <<
 csub3 ;
 }
};

class science : public student //derived class "science"
{
 private:
 char ssub1[20], ssub2[20], ssub3[20] ;
 public:
 void enter_sc(void)
 {
 student_info(); //base class member function
 cout<<"\nEnter the subject1 of the science
 student:";
 cin>>ssub1;
 cout<<"\nEnter the subject2 of the science
 student:";
 cin>>ssub2 ;
 cout<<"\nEnter the subject3 of the science
 student:";
 cin>>ssub3 ;
 }
 void display_sc()
```

```

 {
 display(); //base class member function
 cout<<"\nSubject1 of the science student = " <<
 ssub1 ;
 cout<<"\nSubject2 of the science student = " <<
 ssub2 ;
 cout<<"\nSubject3 of the science student = " <<
 ssub3 ;
 }
};

int main()
{
 arts a ; //a is an object of derived class "arts"
 clrscr();
 cout << "\n Entering details of the arts student\n" ;
 a.enter_arts();
 cout << "\n Displaying the details of the arts student\n" ;
 a.display_arts();
 science s; //s is an object of derived class "science"
 cout << "\n\n Entering details of the science student\n" ;
 s.enter_sc();
 cout << "\n Displaying the details of the science student\n";
 s.display_sc();
 commerce c ; //c is an object of derived class "commerce"
 cout << "\n\n Entering details of the commerce student\n" ;
 c.enter_com();
 cout << "\n Displaying the details of the commerce student\n";
 c.display_com();
 getch();
 return 0;
}

```

---

### 10.4.4 MULTILEVEL INHERITANCE

---

C++ also provides the facility of ***multilevel inheritance***, according

to which the derived class can also be derived by another class, which in turn can further be inherited by another and so on. For instance, a class X serves as a base class for class Y which in turn serves as base class for another class Z. The class Y which forms the link between the classes X and Z is known as the **intermediate base class**. Further, Z can also be used as a base class for another new class. The following figure depicts multilevel inheritance.

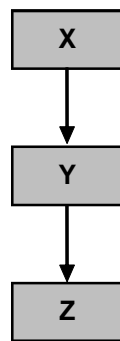


Fig.10.4: Multilevel Inheritance

---

### 10.4.5 HYBRID INHERITANCE

---

It is possible to derive a class involving more than one type of inheritance. **Hybrid inheritance** is that type of inheritance where several forms of inheritance are used to derive a class. There could be situations where we need to apply two or more types of inheritance to design a particular program.

For example, let us assume that we are to design a program which will select players for a particular competition. For this purpose we could consider four classes PLAYER, GAME, RESULT and PHYSIQUE. PLAYER class contains the player details including name, address, location etc. GAME class can be derived from PLAYER class. Again if weightage for physical test should be added before finalizing the result, then we can inherit that from PHYSIQUE. RESULT class is derived from two base classes GAME and

PHYSIQUE. The following diagram gives us the inheritance relationship between various classes.

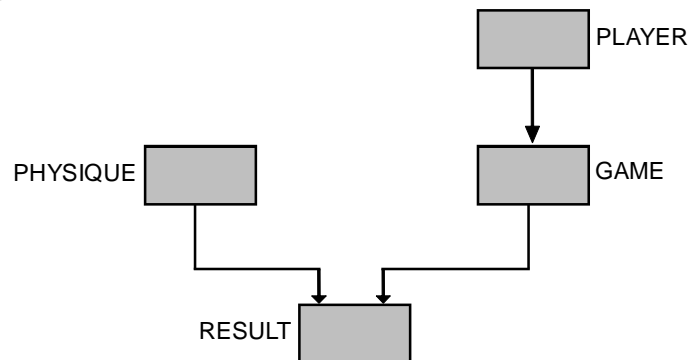


Fig.10.5: Hybrid Inheritance

In the diagram, RESULT has two base classes, GAME and PHYSIQUE. GAME is not only a base class but also a derived class. Here we can see that two types of inheritance *multiple* and *multilevel* are combined to create the RESULT class.

### 10.4.6 MULTIPATH INHERITANCE

The inheritance where a class is derived from two or more classes, which are in turn derived from the same base class is known as ***multipath inheritance***. There may be many types of inheritance such as multiple, multilevel, hierarchical etc. in multipath inheritance. Certain difficulties may arise in this type of inheritance. Suppose we have two derived classes D and E that have a common base class B, and we have another class F that inherits from D and E.

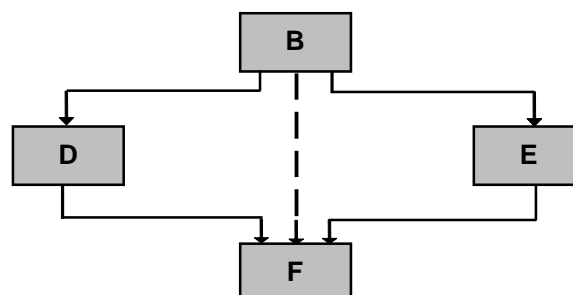


Fig.10.6 : Multipath Inheritance

In the above diagram, we can observe three types of inheritances,



i.e., multiple, multilevel and hierarchical. For better illustration let us consider the following program.

// **Program 10.9:** Demonstration of multipath inheritance

```
#include<iostream.h>
#include<conio.h>
class B
{
 protected:
 int b;
};
class D : public B //B is publicly inherited by D
{
 protected:
 int d;
};
class E : public B //B is publicly inherited by E
{
 protected:
 int e;
};
class F : public D, public E //D,E are publicly inherited by F
{
 protected:
 int f;
 public:
 void enter_number()
 {
 cout<<"Enter some integer values for b,d,e,f: ";
 cin>>b>>d>>e>>f;
 }
 void display()
 {
 cout<<"\nEntered numbers are:\n";
 cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne= "<<e<<"\nf= "<<f;
 }
}
```

```

};
int main()
{
 F obj; //instantiaton of class F
 clrscr();
 obj.enter_number(); //enter_number() of F is called
 obj.display();
 getch();
 return 0;
}

```

Now if we instantiate class F and call the functions **enter\_number()** and **display()**, then the compiler shows the following types error messages:

Error M.cpp 28: Member is ambiguous: 'B ::b' and 'B::b'

Error M.cpp 33: Member is ambiguous: 'B ::b' and 'B::b'

This is due to the duplication of members of class B in F. The member **b** of class B is inherited twice to class F: one through class D and another through class E. This leads to ambiguity. To avoid such type of situation, **virtual base class** is introduced.

---

## 10.5 VIRTUAL BASE CLASSES

---

C++ provides the concept of **virtual base class** to overcome the ambiguity occurred due to *multipath inheritance*. While discussing multipath inheritance, we have faced a situation which may lead to duplication of inherited members in the derived class F (Fig.10.6). This can be avoided by making the common base class (i.e., B) as *virtual base class* while derivation. We can declare the base class B as **virtual** to ensure that D and E share the same data member B.

This is shown in the following program which is the modification of the previous program (i.e., Program 10.9).

/\***Program 10.10:** Virtual base class and removal of ambiguity  
occured in multipath inheritance \*/

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

```
#include<conio.h>
```

```
class B
```

```
{
```

```
 protected:
```

```
 int b;
```

```
};
```

```
class D : public virtual B //B is publicly inherited by D and made virtual
```

```
{
```

```
 protected:
```

```
 int d;
```

```
};
```

```
class E : public virtual B //B is publicly inherited by E and made virtual
```

```
{
```

```
 protected:
```

```
 int e;
```

```
};
```

```
class F : public D, public E //D,E are publicly inherited by F
```

```
{
```

```
 protected:
```

```
 int f;
```

```
 public:
```

```
 void enter_number()
```

```
 {
```

```
 cout<<"Enter some integer values for b,d,e,f: ";
```

```
 cin>>b>>d>>e>>f;
```

```
 }
```

```
 void display()
```

```
 {
```

```
 cout<<"\nEntered numbers are:\n";
```

```
 cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne=
```

```
 "<<e<<"\nf= "<<f;
```

```

 }
};

int main()
{
 F obj; //instantiaton of class F
 clrscr();
 obj.enter_number();
 obj.display();
 getch();
 return 0;
}

```

Here we used the keyword **virtual** in front of the base class specifiers to indicate that only one subobject of type B, shared by class D and class E, exists.. When a class is made a **virtual base class**, C++ takes the necessary action that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

---

## 10.6 ABSTRACT CLASSES

---

The objects created often are the instances of a derived class but not of the base class. The base class just becomes a structure or foundation with the help of which other classes are built and hence such classes are called **abstract class** or **abstract base class**. In other words, when a class is not used for creating objects then it is called **abstract class**. In the Program 10.10 , B is an abstract class since it was not used for creating any object.



### LET US KNOW

#### Inheritance and Constructors, Destructors

Although constructors are suitable for initializing objects , we have not used them in any program in this unit for the sake of simplicity. But if we use constructors in program, then we must follow certain definite rules while inheriting derive classes. If

the base class contains no argument constructor then the de-

derived class does not require a constructor. If any base class contains parameterized constructor, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. In case of inheritance, normally derived classes are used to declare objects. Hence it is necessary to define constructor in the derived class. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in derived class is executed. Destructors are executed in the reverse order of constructor execution.



### CHECK YOUR PROGRESS -2

1. Select whether the following statements are True (T) or False (F):
  - (i) A class can serve as base class for many derived classes.
  - (ii) When one class is derived from another derived class then that is called *multiple inheritance*.
  - (iii) When one class is derived from more than one base class then that is called *multiple inheritance*.
  - (iv) When more than one form of inheritance is used in designing a class then that type is called *hybrid inheritance*.
2. Answer the following by selecting the appropriate option:
  - (i) In multilevel inheritance, the middle class acts as
    - (a) only derived class
    - (b) only base class
    - (c) base class as well as derived class
    - (d) none of the above
  - (ii) A class is declared "virtual" when
    - (a) more than one class is derived
    - (b) two or more classes have common base class
    - (c) there are more than one base classes
    - (d) none of the above

(iii) When a class is not used for creating objects, it is called

- |                    |                        |
|--------------------|------------------------|
| (a) abstract class | (b) virtual base class |
| (c) derived class  | (d) none of these      |

(iv) *Intermediate base class* is present in case of

- |                            |                              |
|----------------------------|------------------------------|
| (a) single inheritance     | (b) multiple inheritance     |
| (c) multilevel inheritance | (d) hierarchical inheritance |

---

## 10.7 LET US SUM UP

---

The key points to be kept in mind in this unit are:

- **Inheritance** is one of the most useful and essential characteristics of object-oriented programming language. If we have developed a class and we want a new class that is almost similar, but slightly different, the principles of inheritance comes handy. The existing class is known as **base** class and the newly formed class is known as **derived** class. The derived class can have some other characteristics which are not in base class.
- Private members of a class cannot be inherited either in public mode or in private mode.
- When a public member inherited in public, protected and private mode, then in derived class it remains with the same access specifiers as in base class i.e., public, protected and private respectively.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in derived class.
- The protected and public data
- In **single inheritance**, one new class is derived from a single base class.
- When a class is derived using the properties of several base classes, then it is called **multiple inheritance**.
- The process of deriving a class from another derived class is called **multilevel inheritance**.

- More than one class can be derived from only one base class i.e., characteristics of one class can be inherited by more than one class. This is called **hierarchical inheritance**.
- When different types of inheritance are applied in a single program then it is termed as **hybrid inheritance**.
- When a class is derived from two or more classes, which are derived from the same base class, such type of inheritance is known as **multipath inheritance**.
- We can make a class **virtual** if it is a base class that has been used by more than one derived class as their base class. When classes are declared as *virtual*, the compiler takes necessary caution to avoid the duplication of the member variables.
- When a class is not used for creating objects, it is called an **abstract class**.



## 10.8 ANSWERS TO CHECK YOUR PROGRESS

### CHECK YOUR PROGRESS-1

1. (i) (b) can be accessed by a derived class  
(ii) (c) private  
(iii) (a) member function of derived class  
(iv) (c) protected  
(v) (b) private  
(vi) (b) only from the base class itself

### CHECK YOUR PROGRESS-2

1. (i) True (ii) False  
(iii) True (iv) True
2. (i) (c) base class as well as derived class  
(ii) (b) two or more classes have common base class  
(iii) (a) abstract class  
(iv) (c) multilevel inheritance



## 10.9 FURTHER READINGS

1. “*Mastering C++*”, by K.R.Venugopal, Rajkumar, Ravishankar, Tata McGraw Hill publication
2. “*The Complete Reference C++*”, by Herbert Schildt, Tata McGraw Hill publication
3. “*Object Oriented Programming with C++*”, by E. Balagurusamy, Tata McGraw Hill publication
4. “*Object-Oriented Programming with ANSI & Turbo C++*”, by Ashok N.Kamthane, Pearson Education



## 10.10 MODEL QUESTIONS

1. What does inheritance mean in C++?
2. What are the types of inheritance? Explain any three of them with examples.
3. What are the different types of visibility modes of base class?
4. Write a program to derive a class from multiple base classes.
5. When do we make a class virtual?
6. What are abstract base classes?
7. Explain multipath inheritance.
8. Write a C++ program involving appropriate type of inheritance which will inherit two classes *triangle* and *rectangle* from *polygon* class. Use member functions for entering appropriate parameters like *width,height* etc. and to calculate the area of triangle and rectangle.
9. Consider a case of University having the disciplines of Engineering, Management, Science, Arts and Commerce. There are many colleges in the University. Assuming a college can run a course pertaining to only one discipline, draw the class diagram. To which type of inheritance this structure belong?



# UNIT-11 : VIRTUAL FUNCTIONS AND POLYMORPHISM

## UNIT STRUCTURE

- 11.1 Learning Objectives
- 11.2 Introduction
- 11.3 Polymorphism
  - 11.3.1 Types of Polymorphism in C++
- 11.4 Virtual Functions
- 11.5 Pure Virtual Functions
- 11.6 Let Us Sum Up
- 11.7 Answers to Check Your Progress
- 11.8 Further Readings
- 11.9 Model Questions

---

### 11.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to–

- learn about polymorphism and its types
- define the rules for virtual function
- use virtual function to achieve run-time polymorphism
- describe and implement pure virtual function

---

### 11.2 INTRODUCTION

---

In the previous unit we have studied the concept of inheritance and its importance in object-oriented programming language like C++.

In this unit we will discuss one useful feature of object-oriented programming, **polymorphism**. It is the ability of objects to take different forms. The ability to display variable behavior depending on the situation is a great

facility in any programming language. In earlier units we have seen operator overloading and overloading of functions. Those are also one kind of polymorphism. C++ supports a mechanism known as **virtual function** to achieve run-time polymorphism. The necessity and usefulness of virtual functions in programming will also be covered in this unit.

---

## 11.3 POLYMORPHISM

---

The word *polymorphism* is a combination of two Greek words, *poly* and *morphism*. *Poly* means *many* and *morphism* means *form*. The functionality of this feature resembles with its name.

---

### 11.3.1 TYPES OF POLYMORPHISM IN C++

---

Polymorphism is supported by C++ both at compile-time and at run-time. Hence, there are two types of polymorphism:

- **Compile-time Polymorphism**
- **Run-time Polymorphism**

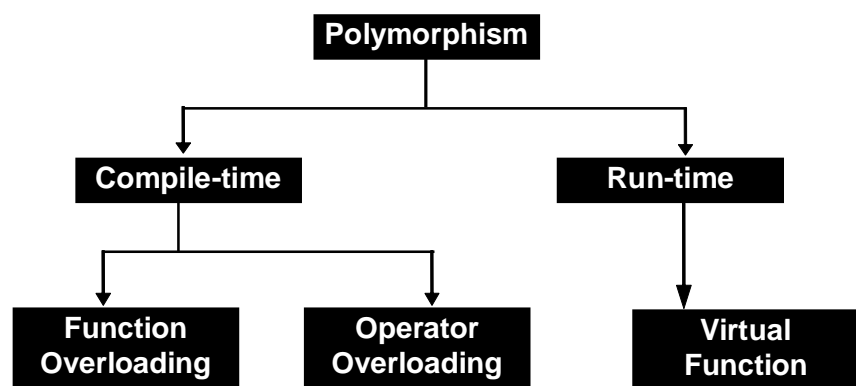


Fig: 11.1: Polymorphism in C++

**Operator overloading** is achieved by allowing operators to operate on the user defined data type with the same manner as that of built-in data types. For example, plus “+” operator produces different actions in case of integers, complex numbers or strings. With the help of

**function overloading**, we can write different functions by using same *function name* but with *different argument lists*. The function would perform different operation depending on the argument list in the function call. The overloaded member functions are selected for invoking by matching the number of arguments and type of arguments. This information is known to the compiler at the compile time itself and therefore the selection of the appropriate function is made at the compile time only.

In both cases, the compiler is aware of the complete information regarding the type and number of operands. Hence, it is possible for the compiler to select a suitable function at compile time. This is known as **compile-time polymorphism**. It is also termed as **static binding** or **early binding**.

Let us consider a program where the function name and argument list are same in both the base and derived class.

**//Program 11.1:**

```
#include<iostream.h>
#include<conio.h>
class B //base class
{
 protected:
 int n;
 public:
 void enter()
 {
 cout<<"Enter a number in base class:\n";
 cin>>n;
 }
 void display()
 {
 cout<<"\nThe number in base class is: "<<n;
 }
}; //end of base class declaration
```

```
class D:public B //derived class D
{
 private:
 int num;
 public:
 void input()
 {
 cout<<"\nEnter a number in derived class:";
 cin>>num;
 }
 void display()
 {
 cout<<"\nThe number in derived class: "<<n;
 }
};

int main()
{
 D d;
 clrscr();
 d.enter(); //will call the enter() of base class
 d.display(); //display() of derived class will be invoked,
 getch();
 return 0;
}
```

Output of the above program will be like this:

Enter a number in base class : 6

The number in derived class : 6

But our intension is to display is :

The number in base class : 6

It has been observed that prototype of **display()** is same in both base and derived class and we cannot term it as *function overloading*. Thus *static binding* does not apply in this case. We have already used statement like **d.B::show()**; in such situation (*program 10.5 of unit Inheritance*); i.e., we used the scope resolution operator (::) to

specify the class while invoking the functions with the derived class objects. But it would be nice if the appropriate member function could be selected while the program is running. With the help of inheritance and virtual functions, C++ determines which version of that function to call. This determination is made at run-time and is known as **run-time polymorphism**. Here the function is linked with a particular class much later after the compilation and thus it is also known as **late binding** or **dynamic binding**. In the following section, we will discuss how to implement virtual function to achieve run-time polymorphism.

---

## 11.4 VIRTUAL FUNCTIONS

---

The concept of virtual functions is different from function overloading. A **virtual function** is a member function that is declared within a base class and redefined by a derived class. The whole function body can be replaced with a new set of implementation in the derived class. To make a function virtual, the **virtual** keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the **virtual** keyword. The difference between a non virtual member function and a virtual member function is, the non virtual member functions are resolved at compile time. Where as the virtual member functions are resolved during run-time.

The concept of *pointers to object* is prior to know before implementing virtual function. We have already studied the concept of pointers in earlier units. At this point, we shall discuss how class members are accessible with the help of pointers.

### Pointers to Objects

A pointer can point to a class object. This is called **object pointer**. Object pointers are useful in creating objects at run time and public members of class can be accessible by object pointers. For example, we can create pointers pointing to classes, as follows:

```
polygon *optr;
```

i.e., class name followed by an asterik (\*) and then the variable name. Thus, in the above declaration, **\*optr** is a pointer to an object of class **polygon**. To refer directly to a member of an object pointed by a pointer we can use arrow operator (->). Here is a program for the illustration of object pointers:

**//Program 11.2:** Demonstration of pointer to object

```
#include<iostream.h>
#include<conio.h>
class polygon
{
 protected:
 int width, height;
 public:
 void set_values (int w, int h)
 {
 width=w;
 height=h;
 }
 void display()
 {
 cout<<"Width : "<<width<<endl<<"Height :
 "<<height;
 }
};
int main ()
{
 polygon p; // p is an object of type polygon
 polygon *optr = &p; // creation and initialization of object pointer
 optr->set_values (8,6); //object pointer accessing member
 optr->display(); //function "set_values()" and "display()"
 getch(); // with arrow operator.
 return 0;
}
```

With the statement      **polygon \*optr = &p;**

we have created object pointer **optr** of type **polygon** and initialized it with the address of **p** object. We can also create the objects using pointers and **new** operator as follows:

```
polygon *optr = new polygon;
```

This statement allocates enough memory for the data members in the object of the particular class and assigns the address of the memory space to **optr**.

### Pointer to base and derived class objects

Pointers can also be used to point base or derived class object. Pointers to object of base class is type-compatible with a pointer to object of derived class. If we create a base class pointer, then that pointer can point to object of base as well as object of derived class.

For example, let us consider the following program :

**//Program 11.3:**

```
#include<iostream.h>
#include<conio.h>
class polygon
{
 protected:
 int width, height;
 public:
 void set_values(int w, int h)
 {
 width=w;
 height=h;
 }
};
class rectangle: public polygon //derived class rectangle
{
 public:
 int area()
 {
 return (width*height);
```

```

 }
 };
 class triangle: public polygon//derived class triangle
 {
 public:
 int area()
 {
 return (width*height / 2);
 }
 };
 int main ()
 {
 rectangle r; // derived class object r
 triangle t; // derived class object t
 clrscr();

 polygon *p1 = &r; //base class pointer pointing derived class object r
 polygon *p2 = &t; // p2 pointing to object t of triangle class

 p1->set_values(5,6);
 p2->set_values(5,6);
 cout<<"\nArea of the rectangle is : "<<r.area()<<endl;
 cout<<"\nArea of the triangle is : "<<t.area()<<endl;
 getch();
 return 0;
 }

```

The output of the program will be like this:

```

Area of the rectangle is : 30
Area of the triangle is : 15

```

In function main, we create two pointers **p1** and **p2** that point to objects of class **polygon**. Then we assign references to **r** and **t** to these pointers. Both are valid assignment operations as because both are objects of classes derived from **polygon**. The only limitation in using **\*p1** and **\*p2** instead of **r** and **t** is that both **\*p1** and **\*p2** are object pointers of type **polygon** and therefore we can only use these pointers to refer to the members that **rectangle** and **triangle** inherit from **polygon**.



The use of pointer to objects of base class with the objects of its derived class does not allow access even to public members of a derived class. That is, it allows access only to those members inherited from the base class but not to the members which are defined to the derived class. For that reason when we call the **area()** members at the end of the program we have had to use directly the objects **r** and **t** instead of the pointers **\*p1** and **\*p2**.

In order to use **area()** with the pointers to base class **polygon**, this member should also have been declared in the class **polygon**, and not only in its derived classes. But the problem is that, **rectangle** and **triangle** implement different versions of **area()**. Therefore we cannot implement it in the base class **polygon**. In such situations, **virtual functions** are necessary.

A pointer to a derived class object may be assigned to a base class pointer, and a **virtual function** called through the pointer. If the function is virtual and occurs both in the base class and in derived classes, then the right function will be picked up based on what the base class pointer really points at.

**//Program 11.3:** Program demonstrating the use of virtual function

```
#include<iostream.h>
#include<conio.h>
class polygon
{
 protected:
 int width, height;
 public:
 void set_values(int w, int h)
 {
 width=w;
 height=h;
 }
 virtual int area() //virtual function
 {
 return (0);
 }
};
```

```

class rectangle: public polygon
{
 public:
 int area() //virtual function redefined
 {
 return (width*height);
 }
};
class triangle: public polygon
{
 public:
 int area() //virtual function redefined
 {
 return (width * height / 2);
 }
};
int main()
{
 rectangle r; //r is an object of derived class rectangle
 triangle t; //t is an object of derived class triangle
 polygon p; //p is an object of base class polygon
 clrscr();
 polygon *p1=&r; //pointer to a derived class object r
 polygon *p2=&t;
 polygon *p3=&p;
 p1->set_values(5,6);
 p2->set_values (5,6);
 p3->set_values (5,6);
 cout<<"Area of the rectangle is: "<<p1->area()<<endl;
 cout<<"Area of the triangle is: "<<p2->area()<<endl;
 cout<<"Area in the polygon class: "<<p3->area()<<endl;
 getch();
 return 0;
}

```

In the above program, the three classes ***polygon***, ***rectangle*** and ***triangle*** have one common member function: ***area()***. The member function ***area()***

has been declared as **virtual** in the base class and it is later redefined in each derived class. The output of the program will be like this:

Area of the rectangle is : 30

Area of the triangle is : 15

Area in the polygon class : 0

If we remove the **virtual** keyword from the declaration of **area()** within *polygon* and run the program, the result will be 0 for the three polygons instead of 30, 15 and 0. That is because instead of calling the corresponding **area()** function for each object (*rectangle::area()*, *triangle::area()* and *polygon::area()*, respectively), **polygon::area()** will be called in all cases since the calls are via a pointer of type **polygon**. A class that declares or inherits a virtual function is called a **polymorphic class**.

When functions are declared as virtual, the compiler adds a data member secretly to the class. This data member is referred to as a **virtual pointer (VPTR)**. A table called **virtual table (VTBL)** contains pointers to all the functions that have been declared as virtual in a class, or any other classes that are inherited. Whenever a call to a virtual function is made in the C++ program, the compiler generates code to treat VPTR as the starting address of an array of pointers to functions. The function call code simply indexes into this array and calls the function located at the indexed addresses. The binding of function call always requires this dynamic indexing activities which always happens at run-time. If a call to a virtual function is made, while treating the object in question, as a member of its base class, the correct derived class function will be called. Thus dynamic binding is achieved with the help of virtual functions.

There are some definite rule for writing virtual function. Those rules are:

- The virtual functions must be members of some class.
- Object pointers should be used to access virtual function.
- A virtual function in a base class must be defined even though it may not be used.
- The prototype of the function which we declare as *virtual* in the base class must be same with all its derived class versions.

- A base pointer can point to any type of the derived object. But we cannot use a pointer to a derived class to access an object of the base class.
- Constructors cannot be virtual but destructors can be virtual.



### CHECK YOUR PROGRESS -1

1. Choose the appropriate option for the correct answer:
  - (i) Run-time polymorphism can be accomplished with the help of
 

|                          |                          |
|--------------------------|--------------------------|
| (a) operator overloading | (b) function overloading |
| (c) virtual function     | (d) friend function      |
  - (ii) Static binding is associated with
 

|                               |                           |
|-------------------------------|---------------------------|
| (a) compile-time polymorphism | (b) run-time polymorphism |
| (c) virtual function          | (d) none of these         |
  - (iii) Pointer to object of base class can point
 

|                       |                          |
|-----------------------|--------------------------|
| (a) base class object | (b) derived class object |
| (c) both (a) and (b)  | (d) none of these        |
  - (iv) Virtual functions can be accessed by
 

|                               |                    |
|-------------------------------|--------------------|
| (a) scope resolution operator | (b) object pointer |
| (c) object                    | (d) none of these  |
  - (v) The ability to take many forms is called .....
 

|                   |                   |
|-------------------|-------------------|
| (a) encapsulation | (b) polymorphism  |
| (c) inheritance   | (d) none of these |
2. State which of the following statements are True (T) or False (F) :
  - (i) The prototype of the function which we declare as *virtual* in the base class must be different with all its derived class versions.
  - (ii) Run-time polymorphism can be achieved only when a virtual function is accessed through a pointer to the base class.
  - (iii) Functions and operator overloading are examples of compile-time polymorphism.

---

## 11.5 PURE VIRTUAL FUNCTIONS

---

Generally, we declare a virtual function inside a base class and redefine it in the derived classes. In many situations there can be no meaningful definition of a virtual function within a base class. Most of the times, the idea behind declaring a function virtual (in the base class), is to stop its execution. Then the question arises why should we define virtual functions? This leads to the idea of pure virtual functions.

For example, in the previous program (*program 11.3*), we have defined a virtual function `area()` within the base class `polygon`. We have also created objects of `polygon` class and made a call to its own `area()` function with object pointer. As the function has minimal functionality, we could leave that `area()` member function without any definition in the base class. This can be done by appending `=0` (equal to zero) to the function declaration as follows:

```
virtual int area() = 0;
```

Such functions are called pure virtual functions. The general form of declaring a pure virtual function is:

```
virtual return_type function_name(parameter_list) = 0;
```

A **pure virtual function** is a virtual function that has no definition within the base class. It only serves as a placeholder. In such cases, the compiler requires each derived class to either define the function or redeclare it as pure virtual function. A class containing pure virtual functions cannot be used to declare objects of its own. Such classes are known as **abstract base class**. As stated earlier, when a class is not used for creating objects then it is called *abstract class* or *abstract base class*, similarly, a class containing pure virtual functions cannot be used for creating objects. A class that cannot instantiate objects is not useless. We can create pointers to it and take advantage of all its polymorphic abilities. Let us examine the working of pure virtual functions with an example:

**//Program 11.4:** Demonstration of pure virtual function

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

```
#include<conio.h>
class polygon
{
 protected:
 int width, height;
 public:
 void set_values(int w, int h)
 {
 width=w;
 height=h;
 }
 virtual int area() = 0; //pure virtual function
};
class rectangle: public polygon
{
 public:
 int area()
 {
 return (width*height);
 }
};
class triangle: public polygon
{
 public:
 int area()
 {
 return (width * height / 2);
 }
};
int main()
{
 rectangle r; //r is an object of derived class rectangle
 triangle t; //t is an object of derived class triangle
 clrscr();
 polygon *p1=&r; //p1 points to object r
 polygon *p2=&t; //p2 points to object t
```

```
p1->set_values(5,6);
p2->set_values (5,6);
cout<<"Area of the rectangle is: "<<p1->area()<<endl;
cout<<"Area of the triangle is: "<<p2->area()<<endl;
getch();
return 0;
}
```

The output will be like this:

Area of the rectangle is : 30

Area of the rectangle is : 15

We can observe that, here we refer to objects of different but related classes using a unique type of pointer (polygon \*p1,\*p2). In the *main()* function, if we try to create object of *polygon* class with statement like **polygon p;** then the compiler will give error message of following type:

Error: Cannot create instance of abstract class 'polygon'.

We should remember that when a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile time error will occur.

### **Virtual function and dynamic allocation of objects**

Virtual member function can also be implemented with dynamically allocated objects. Let us demonstrate the same example with objects that are dynamically allocated.

**/\*Program 11.5:** Demonstration of pure virtual function and dynamically allocated object \*/

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

```
#include<conio.h>
```

```
class polygon
```

```
{
```

```
protected:
```

```
 int width, height;
```

```
public:
```

```
void set_values(int w, int h)
{
 width=w;
 height=h;
}
virtual int area()=0; //pure virtual function
};
class rectangle: public polygon
{
 public:
 int area()
 {
 return (width*height);
 }
};
class triangle: public polygon
{
 public:
 int area()
 {
 return (width * height / 2);
 }
};
int main()
{
 polygon *p1=new rectangle;
 polygon *p2=new triangle;
 clrscr();
 p1->set_values(5,6);
 p2->set_values (5,6);
 cout<<"Area of the rectangle is: "<<p1->area()<<endl;
 cout<<"Area of the triangle is: "<<p2->area()<<endl;
 delete p1;
 delete p2;
 getch();
 return 0;
}
```



In the main() function, we have used the following statements:

```
 polygon * p1= new rectangle;
```

```
 polygon * p2= new triangle;
```

Here the pointer *p1* and *p2* are declared being of type pointer to *polygon* but the objects dynamically allocated have been declared having the derived class type directly.



### CHECK YOUR PROGRESS- 2

1. Choose the appropriate option for the correct answer:
  - (i) Dynamic binding is done using the keyword
    - (a) static
    - (b) dynamic
    - (c) virtual
    - (d) abstract
  - (ii) Virtual function helps us in achieving
    - (a) run-time polymorphism
    - (b) compile-time polymorphism
    - (c) both (a) and (b)
    - (d) none of these
  - (iii) A base class which is not used for object creation is called
    - (a) abstract class
    - (b) derived class
    - (c) virtual class
    - (d) none of these
  - (iv) The function in the statement *virtual show()=0;* is a
    - (a) virtual function
    - (b) pure member function
    - (c) friend function
    - (d) pure virtual function
  - (v) A ..... pointer can point to ..... object
    - (a) derived class, base class
    - (b) void, NULL
    - (c) base class, derived class
    - (d) none of these
2. State which of the following statements are True(T) or False(F):
  - (i) Class containing pure virtual function can instantiate objects of its own.
  - (ii) Pointers to objects of a base class type are compatible with the pointer to objects of a derived class.
  - (iii) A virtual function is a member function that expects to be overridden in a derived class.

---

## 11.6 LET US SUM UP

---

The key points to be kept in mind in this unit are:

- Polymorphism is the ability to use an operator or function in different ways. *Poly*, referring to many, signifies the many uses of these operators and functions. C++ supports polymorphism both at run-time and at compile-time.
- The use of overloaded functions is an example of compile-time polymorphism. Run-time polymorphism can be achieved through the use of pointer to base class and *virtual functions*.
- Object pointers are useful in creating objects at run-time. It can be used to access the public members of an object along with an arrow operator.
- A base class pointer may address an objects of its own class or an object of any class derived from the base class.
- A pure virtual function is a virtual function declared in a base class that has no definition.
- A class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract class or abstract base class.



## 11.7 ANSWERS TO CHECK YOUR PROGRESS

---

### CHECK YOUR PROGRESS -1

1. (i) (c) virtual function      (ii) (a) compile-time polymorphism  
(iii) (c) both (a) and (b)      (iv) (b) object pointer  
(v) (b) polymorphism
2. (i) False      (ii) True      (iii) True

### CHECK YOUR PROGRESS -2

1. (i) (c)virtual      (ii) (a) run-time polymorphism  
(iii) (a)abstract class      (iv) (d)pure virtual function  
(v) (c)base class, derived class
2. (i) False      (ii) True      (iii) True



## 11.8 FURTHER READINGS

1. “*Mastering C++*”, by K.R.Venugopal, Rajkumar, Ravishankar, Tata McGraw Hill publication
2. “*The Complete Reference C++*”, by Herbert Schildt, Tata McGraw Hill Edition
3. “*Object Oriented Programming with C++*”, by E. Balagurusamy, Tata McGraw Hill publication



## 11.9 MODEL QUESTIONS

1. What is polymorphism? What are the different types of polymorphism in C++?
2. How is polymorphism achieved  
(i) at compile time                      (ii) at run-time
3. What is a virtual function?
4. Describe rules for declaring virtual functions.
5. How can C++ achieve dynamic binding?
6. What are pointer to base and derived classes?
7. Write a C++ program to demonstrate the use of abstract classes.
8. Find the error in the following declaration:  

```
class Base{
 public:
 virtual void display()=0;
};
void main() {
 Base b;
}
```
9. What is virtual and pure virtual functions? Use this concept to calculate the area of a square and a rectangle.

## UNIT-12 : FILE HANDLING

### UNIT STRUCTURE

- 12.1 Learning Objectives
- 12.2 Introduction
- 12.3 File Classes
- 12.4 Opening and Closing a File
- 12.5 File Modes
- 12.6 Manipulation of File Pointers
- 12.7 Functions for Input/Output Operations
- 12.8 Let Us Sum Up
- 12.9 Answers to Check Your Progress
- 12.10 Further Readings
- 12.11 Model Questions

---

### 12.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to:

- read data from a file
- write data to a file
- manipulate file pointers
- write programs that perform various operations on files

---

### 12.2 INTRODUCTION

---

In the previous units, while discussing programs we have already used `cout<<, cin>>` functions to write and read data. These functions are console oriented input/output (I/O) functions which always require a keyboard for providing the input and monitor to display the output. This works well when the input/output data is small, but in case the amount of input/output data is substantially large this method has limitations. Also when a program terminates or the computer is turned off, the entire data is lost. So in order

to retain the data a more flexible method has to be used whereby a large amount of data can be stored permanently on disks and read when required. This brings into play the concept of **files** to store and manipulate data easily. In this unit we will be discussing C++ file handling.

---

## 12.3 FILE CLASSES

---

File handling is an important part of all programs. Most of the applications will have their own features to save some data to the local disk and read data from the disk again. C++ file input/output classes simplify such file read/write operations for the programmer by providing easier to use classes. The input/output system of C++ contains a set of classes that define the file handling methods. There are three file input/output classes in C++ which are used for file Read/Write operations.

They are

- ifstream – can be used for File read/input operations.
- ofstream – can be used for File write/output operations.
- fstream – can be used for both read/write C++ file I/O operations.

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: **cin** is an object of class istream and **cout** is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. We also have to #include the <fstream> header file that defines these classes.

---

## 12.4 OPENING AND CLOSING A FILE

---

A file is a place on the disk where a sequence of related data is stored. C++ supports a number of functions that can perform basic file operations like:

- naming a file

- opening a file
- reading data from a file
- writing data into a file
- closing a file

The file name can be a sequence of characters, called as a string. Using the file name a file is recognized. The length of file name depends on the operating system. For example, MS-DOS supports only eight characters as a file name, whereas WINDOWS -98, 2000 or other latest versions supports long file names. A file name contains an extension name also like .doc, .txt, .cpp, .xls, .ppt etc. The following are the valid file names :

|            |                      |
|------------|----------------------|
| Test.cpp   | // extension is .cpp |
| Binary.obj | // extension is .obj |
| Marks.dbf  | // extension is dbf  |

The first step in the file I/O operation is creation of a file stream object and connecting it with the file name. The class ***ifstream***, ***ofstream*** and ***fstream*** can be used for creating file stream defined in the header file ***fstream.h***. The selection of the class is depend on the possible operation(*read/write*) to be performed with the file.

There are two methods of opening a file

- using the member function `open()`
- using constructor of the class

We will concentrate here on the discussion of opening file using the member function **`open()`**.

### **OPENING AND CLOSING A FILE**

The function `open()` is used to open a file. The `open()` function uses the stream object. The `open()` function has two arguments. First argument is *file name* and the second is *mode*. The mode specifies the purpose of opening a file i.e. read, write, append etc. The default values for `ifstream` is `(ios :: in)`, read only and `fstream` is `(ios :: out)`, write only. The file can be

closed explicitly using the close() function.

The syntax for opening and closing file is shown below :

```
file stream class stream object;
stream object . open("file name");
```

For closing a file

```
stream object . close();
```

Examples are shown below :

**A) Opening file for write operation**

```
ofstream out; // create stream object out
out.open("employee.dbf"); // opens file and links with the object out
out.close(); // close the file pointed by the object out
```

**B) Opening file for read operation**

```
ifstream in; // create stream object in
in.open ("employee.dbf"); // opens file and links with the object in
in.close(); // close the file pointed by the object in
```

The following program demonstrates the opening and closing of files as well as writing contents into a file and reading the contents from the file.

**//Program 12.1**

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
 clrscr();
 of stream out;

 //Writing data

 out.open("Week"); // Open file
 out<<"Monday \n"; // Writes string to the file
```

```

out<<"Tuesday \n";
out<<"Wednesday \n";
out.close(); // close the file

//Reading Data

const int N=50;
char text[N];
if stream in;
in.open("Week"); // open file for reading
cout<<"No. of Days : \n";
while(in)
{
 in.getline(text,N);
 cout<<text<<"\n";
}
in.close();
getch();
}

```

---

## 12.5 FILE MODES

---

A file mode is used when opening a file in order to specify the type of operation to be performed with the file. The mode parameter of the **open()** function can have the following flags:

|             |                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| ios::in     | Open the file for input operations.                                                                                                                 |
| ios::out    | Open the file for output operations.                                                                                                                |
| ios::binary | Open the file in binary mode.                                                                                                                       |
| ios::ate    | Set the initial position at the end of the file. If this flag is not set to any value, 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. This flag can only be used in |



|                |                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------|
|                | streams open for output-only operations.                                                                                      |
| ios::trunk     | If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one. |
| ios::nocreate  | do not create the file, open only if it exists.                                                                               |
| ios::noreplace | open and create a new file if the specified file does not exist.                                                              |

All these flags can also 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 one of the `open()` member functions of the 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     |

For *ifstream* and *ofstream* classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

---

## 12.6 MANIPULATION OF FILE POINTERS

---

The C++ input and output system manages two integer values associates with a file. These are:

- Input or get pointer – specifies the location in a file where the next read operation will occur.
- Output or put pointer – specifies the location in a file where the next write operation will occur.

In other words, these pointers indicate the current positions for read and write operations, respectively. Each time an input or an output operation takes place, the pointers automatically advance sequentially. We can use these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

***The term pointers should not be confused with normal C++ pointers used as address variables.***

By default the reading pointer and the writing pointer are set at the beginning and at the end (when you open file in `ios::app` mode) of a file respectively. There are times when we must take control of the file pointers ourselves so that we can read from and write to any arbitrary location in the file. There are four functions that allow us to do so. They are:

| Function             | Description                                       |
|----------------------|---------------------------------------------------|
| <code>seekg()</code> | Moves get pointer (input) to a specified location |
| <code>seekp()</code> | Moves put pointer(output) to a specified location |
| <code>tellg()</code> | Gives the current position of the get pointer     |
| <code>tellp()</code> | Gives the current position of the put pointer     |

In other words, these four functions allow us to access a file in a non-sequential or random mode.

### **Using seekg() with one argument**

General syntax for using `seekg()`:

`fl1.seekg(k)`

where k is absolute position from the beginning. The start of the file is byte 0.

**Example:** *ifstream pos;*

```
pos.seekg(10);
```

```
// means, "position the get pointer 10 bytes from the beginning
of the file"
```

### **Using seekp() with one argument**

General syntax for using seekp() with one argument is:

```
fl1.seekp(k)
```

where k is absolute position from the beginning. The start of the file is byte 0.

**Example:** *ifstream pos;*

```
pos.seekp(10);
```

```
// means, "position the put pointer 10 bytes from the beginning
of the file"
```

The functions seekg() and seekp() can also be used with two arguments by specifying the offset.

General syntax is:

```
seekg(offset, refposition);
```

```
seekp(offset, refposition);
```

The first argument "offset" is an integer that specifies the number of byte positions(also called offset). The second argument "refposition" is the reference point. The table below shows some of the sample pointer offset calls and their actions.

| Seek call                       | Action                       |
|---------------------------------|------------------------------|
| <code>seekg(0,ios::beg);</code> | Go to the start              |
| <code>seekg(0,ios::cur);</code> | Stay at the current position |

|                                   |                                                  |
|-----------------------------------|--------------------------------------------------|
| <code>seekg(0,ios::end);</code>   | Go to the end of the file                        |
| <code>seekg(m,ios::beg);</code>   | Move to (m+1 ) th byte in the file               |
| <code>seekg(m,ios::cur);</code>   | Go forward by m bytes from the current position  |
| <code>seekg(-m, ios::cur);</code> | Go backward by m bytes from the current position |
| <code>seekg(-m, ios::end)</code>  | Go backward by m bytes from the end              |

A negative value (-m) moves the file pointer backwards from the reposition.

### **Using tellg() and tellp()**

The *tellg()* and *tellp()* functions can be used to find out the current position of the get and put file pointers respectively in a file.

These two member functions have no parameters and return a value of the member type *pos\_type*, which is an integer data type representing the current position of the get stream pointer (in the case of *tellg*) or the put stream pointer (in the case of *tellp*).

The following program demonstrates the use of the *seekp()* and the *tellp()* functions.

#### **// Program 12.2**

```
#include<iostream.h>
#include<fstream.h>

void main()
{
 long pos;
 ofstream outfile;
 outfile.open ("test.txt");
 outfile.write ("This is an apple",16);
 pos=outfile.tellp();
 outfile.seekp (pos-7);
 outfile.write (" sam",4);
 outfile.close();
}
```

In this example, *tellp* is used to get the position of the put pointer after the writing operation. The pointer is then moved back 7 characters to modify the file at that position, so the final content of the file shall be:

This is a sample

---

## 12.7 FUNCTIONS FOR INPUT/OUTPUT OPERATIONS

---

There are number of functions to perform read and write operations with the files. Some function read/write single character and some function read/write block of binary data. The ***put()*** and ***get()*** functions are used to read or write a single character whereas ***write()*** and ***read()*** are used to read or write block of binary data.

### The ***put()*** and ***get()*** function :

The ***get()*** function is a member function of the class *fstream*. This function reads a single character from the file pointed by the get pointer i.e. the character at current get pointer position is caught by the ***get()*** function.

The ***put()*** function writes a character to the specified file by the stream object. It is also a member of the *fstream* class. The ***put()*** function places a character in the file indicated by put pointer.

The following program demonstrates a read and write string to a file using the ***get()*** and ***put()*** function.

### // Program 12.3

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
 clrscr();
 char str[50];
 cout<<"Enter a String :";
```

```

 cin.getline(str,50);
 int i=0;
 fstream in;
 in.open("test", ios::in | ios::out);
 while(str[i] !='\0')
 in.put(str[i++]); // Writes characters to the file
 in.seekg(0); // Set the file pointer at beginning
 char c;
 cout<<"\n Entered String :";
 while(in)
 {
 in.get(c); // Reads a character
 cout<<c;
 }
 getch();
 }

```

### **write() and read() function :**

The data entered by the user are represented in ASCII format. But the computer understands only the binary format i.e. 0 or 1. When data are stored in text format the numbers are stored as characters and occupies more memory space. The functions *put()* and *get()* read/write a character. The data is stored in the file in character format. If large number of numeric data is stored in the file, it will occupy more space. Hence, using *put()* and *get()* creates disadvantages.

Using the *read()* and *write()* function this problem can be eliminated because these functions handles only binary format of data. In binary format, the data representation in the file and in the system is same. Remember that ASCII format of data always has to be converted into binary format for processing by CPU.

The format of the *write()* and *read()* functions are given below :

**in.read((char \*) &p, sizeof(p));**

---

```
out.write((char *) &p, sizeof(p));
```

There are two arguments in these functions. The first argument is the address of the variable p. The second argument is the size of the variable p (in bytes).

The following program shows the use of the read() and write() function by creating binary files.

#### **// Program 12.4**

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
 clrscr();
 float digit[]={5.25,6.25,7.25};
 //Writing data into file
 ofstream out;
 out.open("test.bin"); // file test.bin is opened
 out.write((char *) &digit, sizeof(digit)); // writes into the file
 out.close();
 for(int i=0; i<3; i++)
 digit[i]=0;

 //Reading data from file
 ifstream in;
 in.open("test.bin");
 in.read((char *) &digit, sizeof(digit));
 for(i=0; i<3; i++)
 cout<<digit[i]<<"\t";
 getch();
}
```



## CHECK YOUR PROGRESS

### 1. Fill in the blanks:

- (a) We use \_\_\_\_\_ to store huge amount of data.
- (b) The \_\_\_\_\_ file class can be used for file write/output operations
- (c) The \_\_\_\_\_ function is used to open a file.
- (d) `ios::in` opens a file for \_\_\_\_\_ operations.
- (e) The default mode parameter of the `ifstream` class is \_\_\_\_\_.
- (f) The \_\_\_\_\_ function moves the put pointer to a specified location.

### 2. State true or false.

- (a) The `ifstream` class is used for file I/O operations.
- (b) The mode parameter `ios::binary` opens a file for input operations.
- (c) The `tellp()` pointer gives the current position of the get pointer.
- (d) The functions `seekg()` and `seekp()` can only be used with one argument.
- (e) The `seekp()` function moves the put pointer to a specified location.

---

## 12.8 LET US SUM UP

---

- Files are used to store huge collection of data permanently. The stored data can later be used by performing various file operations like Opening, Reading, Writing etc.
- C++ File I/O **classes** provide a easier way to perform I/O operation on files. These classes define the file handling methods. They are **`ifstream`**, **`ofstream`** and **`fstream`**.



- To open a file we use the **open()** function with parameters specifying the file to open and the mode in which it is to be opened. It can be closed with the **close()** function.
- **File modes** specify the type of operations to be performed with the opened file. The mode parameter of the **open()** function can have eight different values to specify the kind of operation to be performed.
- The **get** and **put** pointers indicate the current positions for read and write operations.
- There are four functions allow us to access a file in a non-sequential or random mode. They are **seekg()**, **seekp()**, **tellg()** and **tellp()**.
- The **seekg()** and **tellg()** functions allow us to set and examine the get pointer, and the **seekp()** and **tellp()** functions perform these same actions on the put pointer. The **seekg()** and **seekp()** functions can be used with one argument as well as two arguments. The **tellg()** and **tellp()** functions have no arguments.



## 12.9 ANSWERS TO CHECK YOUR PROGRESS

1. (a) files                      (b) ofstream                      (c) open()  
(d) input                      (e) ios::in                      (f) seekp().
2. (a) True.  
(b) False, since it opens a file in binary mode.  
(c) False, since it gives the current position of the put pointer.  
(d) False, since seekg() and seekp() can be used with two arguments by specifying the offset.



## 12.10 FURTHER READINGS

1. John R. Hubbard: Programming with C++, Tata McGraw-Hill publication.
2. E. Balagurusamy: Programming with C++, Tata McGraw-Hill publication.



## 12.11 MODEL QUESTIONS

---

1. What are file classes? State their functions.
2. How many file modes can be used with the `open()` function to open a file? State the function of each mode.
3. What do the `seekg()` and `seekp()` functions do?
4. Explain `tellg()` and `tellp()`.
5. How can we use `seekg()` and `seekp()` with two arguments.
6. Explain the use of input/output functions `put()`, `get()`, `write()` and `read()` with an example each.
7. Write a program to find out the number of records in the file `billfile.dat` by using the `seekg()` and `tellg()` functions.
8. Write a program that writes a structure to disk and then reads it back in.