KEC

New Edition

Object Oriented Programming

B.Sc. CSIT Second Semester

Arjun Singh Saud

Table of Contents

CHAPTER 1

Overview of Object Oriented Programming (OOP)

CHAPTER 2

Basics of C++ Programming

CHAPTER 3

Control Structures

CHAPTER 4

Pointers and Functions

CHAPTER 5

Classes and Objects

CHAPTER 6

Constructors and Destructors

CHAPTER 7

Operator Overloading

CHAPTER 8

Inheritance

CHAPTER 9

Dynamic Polymorphism

CHAPTER 10

Exception Handling

CHAPTER 11

Templates

CHAPTER 12

Input/Output with Files

Chapter 1

Introduction of Object Oriented Programming

Objectives

After studying this unit, you will be able to:

- Recognize the basic concept of object-oriented programming.
- Describe the OOP languages.
- Compare the procedural Oriented and object-oriented programming.

1.1 PROGRAMMING LANGUAGES

As a loose definition, a programming language is a tool used by a programmer to give the computer very specific instructions in order to serve some purpose for the user. A program is like a recipe. It outlines exactly the steps needed to create something or perform a certain task. For, example, when baking chocolate chip cookies, there are certain steps that need to be followed:

```
Mix eggs, butter, sugar in a bowl
Add flour, baking soda, and flavorings
Mix until creamy
Add chocolate chips
Bake in the oven
```

For a person who has made cookies before and knows the amounts of each ingredient to use, this recipe is sufficient, however, for a person who has never baked cookies before, this recipe will not do. That person would need a recipe like the following:

```
Place two eggs in a bowl
Add 1.5 c. butter to the eggs
Bake cookies for 10-12 minutes at 375 degrees or until brown
```

There is still a problem with the preceding recipe. The first instruction says to put two eggs in the bowl, but it doesn't say to shell them first! This may seem like common sense. Computers do exactly what they are told, no more, no less. When writing a program, a programmer must outline every possible step and scenario that could occur.

The first programming languages that emerged were assembly languages. These languages are exactly the instruction set of a specific processor. These languages are very low-level and hard to understand. For example, say we wanted to add two numbers, 3 and 4 and get a result:

```
In C++                    In Assembly
int a = 3+ 4              Load 3 R1
                          Load 4 R2
                          Add R1 R2 R3
```

This is analogous to the differences in the first and second recipe presented. The first recipe expressed the method of baking cookies on a high level, while the second method went more in depth on how to actually mix and bake the cookies. Programmers write their code in a high level language and then use a compiler to translate their code into an assembly language and then into a machine language that will run on the machine they are using.

Programs consist of algorithms. An algorithm is just a well-outlined method for completing a task. The above recipes could be called algorithms for the task of baking cookies. A high level algorithm for adding two numbers could be as follows:

```
Ask the user for the first number
Ask the user for the second number
Add the two numbers
Display the result on the screen
```

This high-level abstraction is not actual code. However, it does express the ideas of a program, and is called pseudo-code. Often, programmers will design their programs in pseudo-code, and then use this to write their actual code.

So, why is there more than one programming language? It may seem that a standard language should be agreed on, since all languages are translated using a compiler anyways. However, languages are often designed with a specific use in mind, and some are better than others for dealing with certain problems. So if a programmer is capable of writing a compiler (which is a very complex piece of software) then they can design and create a language.

The most important thing to remember about programming languages is that they are only an abstraction! Programming languages were created so developers could express their ideas on a higher level than a computer can understand. Once a user has a good concept of how computers work, and has learned a few computer languages, it becomes much easier to pick up new languages.

A programming language is a tool used by programmers in order to specifically outline a series of steps that a computer is to take in a certain instance. High-level programming languages allow a programmer to express ideas on an abstract level, and force the compiler to worry about the low-level implementation details. This allows for faster development of applications, since applications are easier to write. There are even fourth generation languages emerging as viable programming languages. Recall that machine code is considered first generation, assembly languages are second generation, and compiled languages are third generation. Fourth generation languages are actually code-generating environments, such as Microsoft's Visual Basic. These fourth generation languages allow programmers to express their ideas visually, and the environment then writes the code to implement these ideas.

1.2 A SURVEY OF PROGRAMMING TECHNIQUES

Roughly speaking we can distinguish the following learning curve of someone who learns program.

- Unstructured programming
- Structured programming and
- Object oriented programming

1.2.1 Unstructured Programming

This is the programming approach where programmers put all their code in one place or say in one program. There is no harm as such but when line of code increases, situation starts getting unmanageable. This programming technique provides tremendous disadvantages once the program gets sufficiently large. For example if the same statement sequence is needed at different locations within the program the sequence must be copied. A programmer can face so many problems like:

- Duplicacy and redundancy of code
- Code maintenance problem
- Readability and many more

Thus, this approach is totally useless if working on a large project. It can be only helpful while testing one or two features etc.

1.2.2 Structured Programming

To overcome above situation it was needed to come up with a Structured programming approach to make some code separation keeping reusability in mind. In this approach a new idea came up and a set of execution code was kept in a place and it was called function or procedure. A procedure call is used to invoke the procedure. By introducing parameters as well as procedures of procedures programs can now be written more structured and error free manner. For example if a procedure is correct every time it is used it produces correct results. Consequently in cases of errors you can narrow your search to those places which are not proven to be correct. Now a program can be viewed as a sequence of procedure call. The main program is responsible to pass data to the individual calls. The data is processed by the procedures and once the program has finished the resulting data is presented.

The basic feature of Procedural-oriented Programming is to reuse the same code at different places in the program without copying it. In this technique, program flow follows a simple hierarchical model that employs three types of control flows: *sequential, selection, and iteration.* Most modern procedural languages include features that encourage structured programming. Some of the better known structured programming languages are Pascal, C, Ada etc.

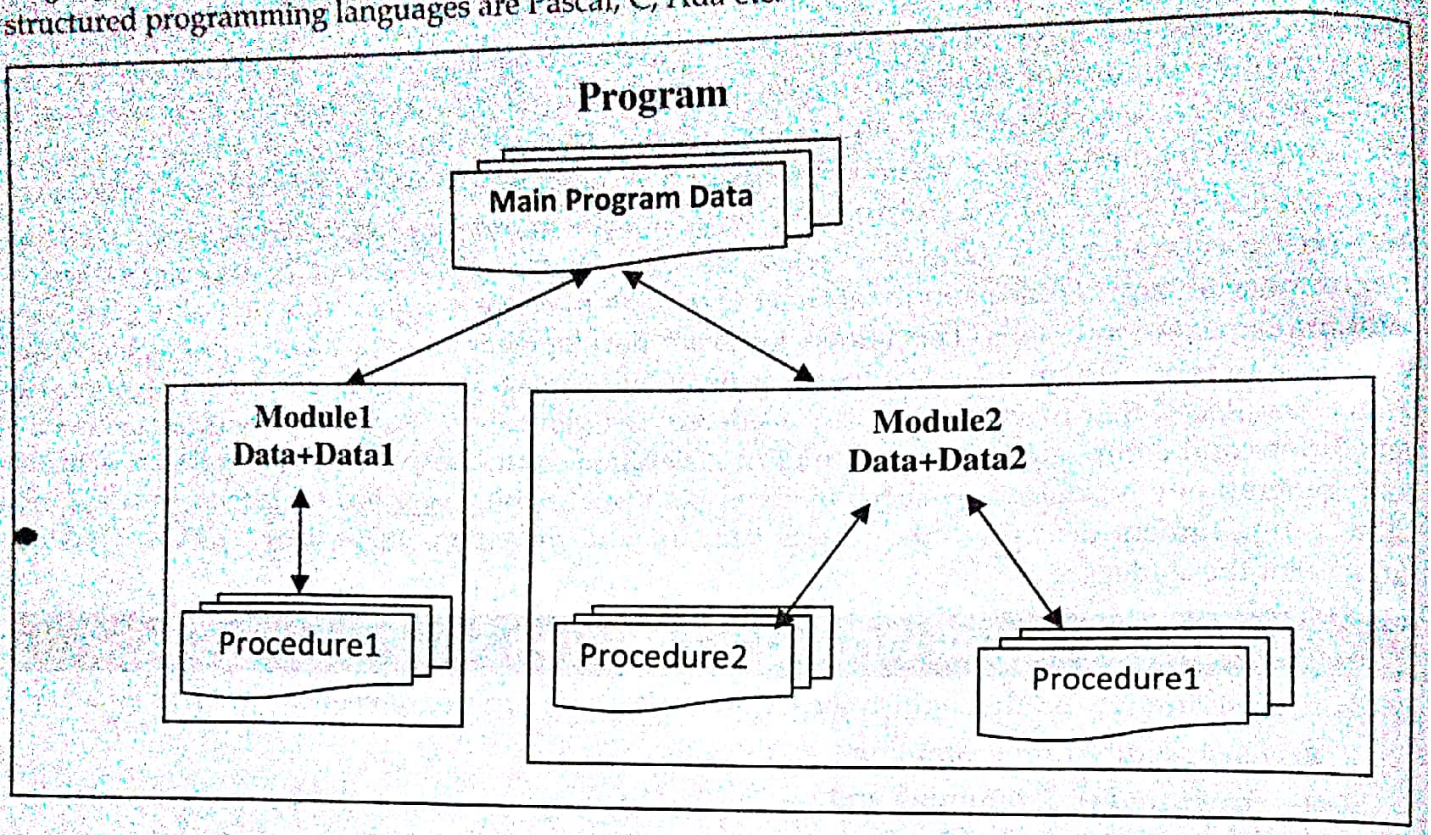


Figure 1.1: Program Structure of Structured Programming.

Some characteristics exhibited by structured programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function

Some problems with structured programming are:

- **Complexity of Managing Large Projects:** As programs grow larger, even structured programming approach begins to show signs of strain. No matter how well the structured programming approach is implemented, the project becomes too complex, the schedule slips, more programmers are needed, and costs skyrocket.
- **Data Undervalued:** Data is given second-class status in the organization of structured languages. A global data can be corrupted by functions. Since many functions access the same global data, the way the data is stored becomes critical.
- **Relationship to the Real World:** Procedural programs are often difficult to design because their chief components – functions and data structures – don't model the real world very well.
- **New Data Types:** It is difficult to create new data types with procedural languages. Furthermore, most Procedural languages are not usually extensible and hence procedural programs are more complex to write and maintain.

1.2.3 Object-oriented Programming

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model. An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called sending a message to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called message. In addition, object-oriented programming supports encapsulation, abstraction, inheritance, and polymorphism to write programs efficiently. Examples of object-oriented languages include Simula, Smalltalk, C++, Python, C#, Visual Basic .NET, Java etc. Simula was the first object-oriented programming language.

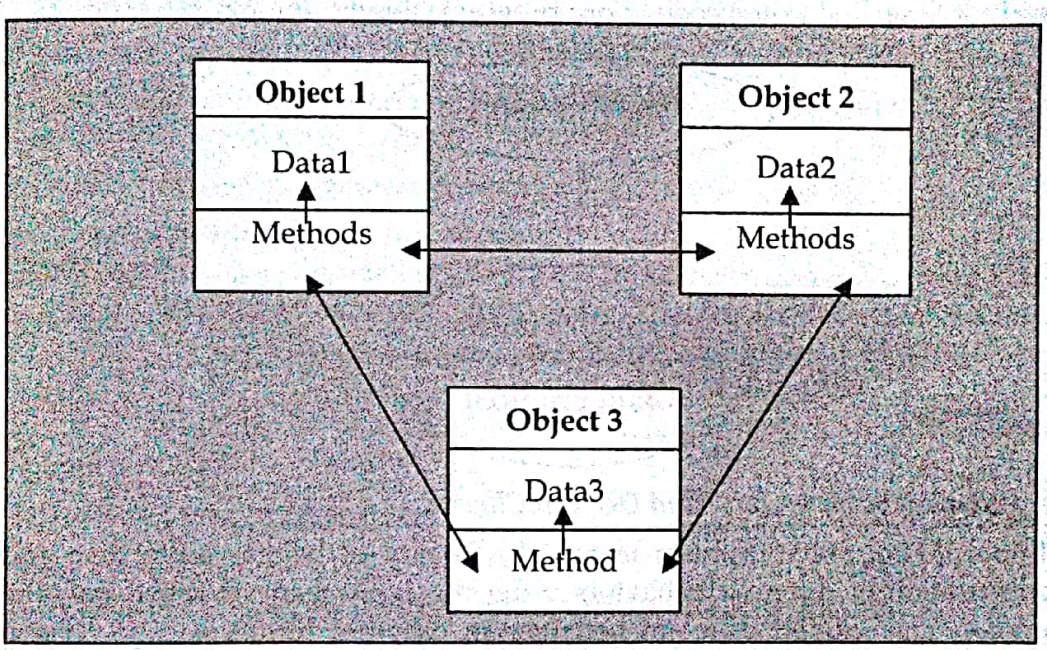


Figure 1.2: Interaction between objects.

Characteristics of OOP

The basic concepts underlying OOP are: *Class, Object, abstraction, encapsulation, inheritance, polymorphism, and Message Passing.*

- **Objects:** Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user defined data such as vectors, time and lists. They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data.
- **Class:** A class represents a set of related objects. The object has some attributes, whose value consist much of the state of an object. The class of an object defines what attributes an object has. The entire set of data and code of an object can be made a user-defined data type with the help of a class. Thus classes are user defined data types that behave like the built-in types of a programming language. Classes have an interface that defines which part of an object of a class can be accessed from outside and how, a class body that

implements the operations in the interface, and the instance variables that contain the state of an object of that class.

- **Abstraction:** Abstraction is the essence of OOP. Abstraction means the representation of the essential features without providing the internal details and complexities. In OOP, abstraction is achieved by the help of class, where data and methods are combined to extract the essential features only.
- **Encapsulation:** Encapsulation is the process of combining the data (called fields or attributes) and functions (called methods or behaviors) into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privileges to the data inside the class. **Data hiding** is possible due to the concept of encapsulation, since the data are hidden from the outside world.
- **Inheritance:** Inheritance is the process of acquiring certain attributes and behaviors from parents. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class (called derived class) from another class (called base class), some of the data and functions can be inherited so that we can reuse the already written and tested code in our program, simplifying our program.

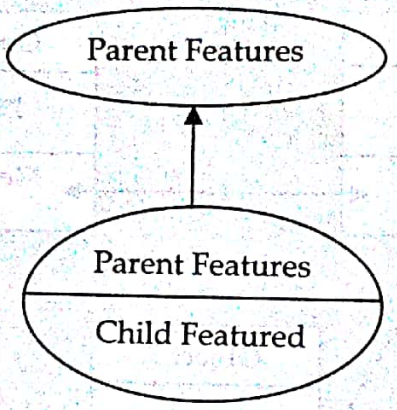


Figure 1.3: Relationship between Base and Derived Class

- **Polymorphism:** Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. An operation may exhibit different behavior in different instances. The behavior depends upon the types of the data used in the operation. For example considering the operator plus (+).

  16 + 4 = 20

  "TU" + ",CDCSIT" = "TU,CDCSIT"

  However the behavior depends upon the attribute the name holds at particular moment.

  Dynamic Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of call at run-time. This is associated with polymorphism and inheritance. Function call associated with a polymorphic reference depends on the dynamic type of that reference. At run-time, the code matching the object under reference will be called.
- **Message Passing:** Message passing is another feature of object-oriented programming. An object-oriented program consists of a set of objects that communicate with each other. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts. A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that

generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information (Arguments) to be sent.

Benefits of OOP

Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The technology provides greater programmer productivity, better quality of software and lesser maintenance cost. The principle advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding help the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Object-based Programming

Object oriented programming is not the right of any particular language. Although languages like C and Pascal can be used but programming becomes clumsy and may generate confusion when program grow in size. A language that is specially designed to support the OOP concepts makes it easier to implement them. To claim that they are object-oriented they should support several concepts of OOP as stated above. Depending upon the features they support, they are classified into the following categories:

- Object-based programming languages.
- Object-oriented programming languages.

The fundamental idea behind object-based programming is the concept of objects (the idea of encapsulating data and operations) where one or more of the following restrictions apply:

- There is no implicit inheritance
- There is no polymorphism
- Only a very reduced subset of the available values are objects (typically the GUI components)

Object-based languages need not support inheritance or sub-typing, but those that do are also said to be object-oriented. An example of a language that is object-based but not object-oriented is Visual Basic (VB). VB supports both objects and classes, but not inheritance, so it does not qualify as object-oriented. Sometimes the term object-based is applied to prototype-based languages, true object-oriented languages that do not have classes, but in which objects instead inherit their code and data directly from other template objects. An example of a commonly used prototype-based language is JavaScript.

1.3 SOFTWARE CRISIS

Exactly what gave birth to object-oriented approach in programming couldn't be stated clearly. However, it is almost certain that by the end of last decade millions and millions lines of codes have been designed and implemented all over the world. The colossal efforts that went into developing

this large population of programs would be rendered useless if these codes could not be reused. Programmers were, most of time, busy creating perhaps the same thing again and again; writing same or similar codes repeatedly in each project implementation.

Software industry faced a set of problems that are encountered in the development of computer software other than programs not functioning properly. Surveys conducted time to time revealed that more and more software development projects were failing to come to successful completion. Schedule and cost overrun were more frequent than seemed to be. This situation has been dubbed by various industry experts as "software crisis". In the wake of this, questions like – how to develop software, how to support a growing volume of existing software, and how to keep pace with a rapidly growing demand for more error-free and efficient software – became important concerns in connection with software development activities.

Soon software crisis became a fact of life. Engineers and Scientists of the computing community were forced to look into the possible causes and to suggest alternative ways to avoid the adverse affects of this software crisis looming large at software industry all over the world. It is this immediate crisis that necessitated the development of a new approach to program designing that would enhance the reusability of the existing codes at the least. This approach has been aptly termed as object-oriented approach.

1.4 A BRIEF HISTORY OF C++

The C++ Programming Language is basically an extension of the C Programming Language. The C Programming language was developed from 1969-1973 at Bell labs, at the same time the UNIX operating system was being developed there. C was a direct descendant of the language B, which was developed by Ken Thompson as a systems programming language for the fledgling UNIX operating system. B, in turn, descended from the language BCPL which was designed in the 1960s by Martin Richards while at MIT.

In 1971 Dennis Ritchie at Bell Labs extended the B language (by adding types) into what he called NB, for "New B". Ritchie credits some of his changes to language constructs found in Algol68, although he states "although it [the type scheme], perhaps, did not emerge in a form that Algol's adherents would approve of" After restructuring the language and rewriting the compiler for B, Ritchie gave his new language a name: "C".

In 1983, with various versions of C floating around the computer world, ANSI established a committee that eventually published a standard for C in 1989. In 1983 Bjarne Stroustrup at Bell Labs created C++. C++ was designed for the UNIX system environment, it represents an enhancement of the C programming language and enables programmers to improve the quality of code produced, thus making reusable code easier to write.

Self Assessment

Fill in the Blanks.

1. Each object contains data and ..... to manipulate the data.
2. ..... have an interface that defines which part of an object of a class can be accessed from outside and how.
3. The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the ..... class.
4. At run-time, the code matching the ..... under reference will be called.
5. .....programming focuses on algorithm rather than data.

6. Languages that support programming with objects are said to be ..... programming languages.
7. The colossal efforts that went into developing this large population of programs would be rendered useless if these codes could not be.....
8. Objects communicate with each other by using.....
9. Object oriented programming can eliminate ..... code and extend the use of existing classes.
10. It is possible to have ..... instances of an object to co-exist without any interference.

Exercise

1. Discuss the working of procedure-oriented programming with suitable block diagram of program execution.
2. How procedure oriented programming is different from structured programming? Discuss.
3. Inheritance is the process by which objects of one class acquire the properties of objects of another class. Analyze.
4. Why data is considered insecure in procedure oriented programming paradigm? Explain with examples.
5. Examine what are the benefits of object oriented programming over structured programming?
6. How are classes different from objects? Illustrate with suitable examples.
7. What supports are necessary for a programming language to classify it as an object oriented programming language?
8. The technology provides greater programmer productivity, better quality of software and lesser maintenance cost. Explain how?
9. Scrutinize the meaning of dynamic binding with an example.
10. Make distinctions between data abstraction and data encapsulation. Give examples.

❑❑❑

Chapter 2

Basics of C++ Programming

Objectives

After studying this unit, you will be able to:

- Recognize the tokens.
- Describe the expressions.
- Explain the operators.
- Explain the scope resolution operator.
- Recognize the reference variable.

2.1 CHARACTER SET OF C++

Character set is a set of valid characters that a language can recognize. A character represents any letter, digits, or any other sign. C++ has the following character set:

- **Letters:** Letters refer to the 26 alphabets that used in the English language. Both uppercase (A-Z) and lowercase (a-z) alphabets can be used in C++.
- **Digits:** Digits from 0-9 or any combination of these can be used in any C++ code.
- **Special Characters/Symbols:** Apart from the usual letters and digits, we have a set of special characters that can be used in C++. All these characters are used for various purposes. These are:

```
+ - * / ^
\ () [] {}
= != < > <= >=
' " $ , ; :
% ! & ?
_ # @
Space
```

- **White Spaces:** These are a special set of characters that are mainly used for the purpose of formatting the text in the C++ program. They are also called formatting characters. Some common examples are blank space, horizontal tab, newline, carriage return, vertical tab, form feed etc.

2.2 TOKENS

The smallest individual units in a program are known as tokens. C++ has the following tokens:

1. Keywords
2. Identifiers
3. Constants
4. Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications. Tokens are usually separated by white space. White space can be one or more: *Blanks, Horizontal or vertical tabs, New lines, and Form feeds.*

2.2.1 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Table 2.1 gives the complete set of C++ keywords. The keywords not found in ANSI C are shown in boldface. These keywords have been added to the ANSI C keywords in order to enhance its features making it an object-oriented language.

asm	double	new	switch
auto	else	**operator**	**template**
break	enum	**private**	This
case	extern	**protected**	**Throw**
catch	float	**public**	**Try**
char	For	register	typedef

class	**friend**	return	Union
const	goto	short	unsigned
continue	If	signed	**virtual**
Default	**inline**	sizeof	Void
delete	int	static	volatile
do	long	struct	while

Table 2.1: Keywords.

2.2.2 Identifiers

Identifiers refer to the names of variables, functions, arrays, classes etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are not distinct.
- A reserved keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and, therefore, all the characters in a name are significant. Care should be exercised while naming a variable that is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

2.2.3 Constants

A constant is any expression that has a fixed value. They can be divided in Integer Numbers, Floating-Point Numbers, Characters and Strings.

Integer Numbers

1776
707
-273

they are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write **1776** in a program we will be referring to the value 1776. In addition to decimal numbers C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we must precede it with a **0** character (zero character). And to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

All of them represent the same number: 75 (seventy five) expressed as a decimal number, octal and hexadecimal, respectively.

Floating Point Numbers

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number 3 expressed as a floating point numeric literal. There also exist non-numerical constants, like:

Non-Numeric Constants

'z'
'p'
"Hello world"
"How do you do?"

The first two expressions represent single characters, and the following two represent strings of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string of more than one character we enclose them between double quotes ("). When writing both single characters and strings of characters in a constant way, it is necessary to put the quotation marks to distinguish them from possible variable identifiers or reserved words. Notice this:

x and 'x'

x refers to variable x, whereas 'x' refers to the character constant 'x'. Escape characters are special characters that cannot be expressed otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by an inverted slash (\). Here is the list of such escape codes:

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	page feed
\a	alert (beep)
\'	single quotes (')

2.2.4 Operators

Operators are symbols or combination of symbols that directs the computer to perform some operation upon operands. C++ includes a large number of operators, which fall into several different categories. In this section we examine some of these categories in detail. Specifically, we will see how arithmetic operators; unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions. The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Operators are used to compute and compare values, and test multiple conditions. They can be classified as:

1. Arithmetic operators
2. Assignment operators
3. Unary operators
4. Comparison operators
5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

Arithmetic Operators

There are five arithmetic operators in C++. Their name and description is listed in the table given below:

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	remainder after integer division

Table 2.2: Arithmetic Operators with their function

The % operator is sometimes referred to as the modulus operator. There is no exponentiation operator in C++. However, there is a library function (pow) to carry out exponentiation. Alternatively you can write your own function to compute exponential value. The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set.). The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another integer is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-pointing quotient. Suppose that 'a' and 'b' are integer variables whose values are 8 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	12
a – b	4
a * b	32
a / b	2
a % b	0

Table 2.3:
Arithmetic Expressions and their values.

Now suppose that a1 and a2 are floating-point variables whose values are 14.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a1 + a2	16.5
a1 – a2	12.5
a1 * a2	29.0
a1 / a2	7.25

Table 2.4: Floating point Arithmetic.

Finally, suppose that x1 and x2 are character-type variables that represent the character M and U, respectively. Some arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

x1+x2 = 162

x1+x2+'5'=215

Note that M is encoded as (decimal) 77, U is encoded as 85, and 5 is encoded as 53 in the ASCII character set. If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient. The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C++ assign the sign of the first operand to the remainder.

Assignment Operators

Assignment operator assigns the value of right operand or expression to the variable in left side. There are many variations of assignment operator as described below

Operator	Description	Example	Explanation
=	Assign the value of the right operand to the left	a = b	Assigns the value of b to a
+=	Adds the operands and assigns the result to the left operand	a +=b	Adds the value of b to a and assign the result to a. The expression could also be written as a = a+b
-=	Subtracts the operand in right side from operand in left side and assigns the result to the left operand	a -=b	Subtracts value of 'b' from value of 'a' and assign the result to 'a'. The expression could also be written as a = a-b
*=	Multiplies the operands and assigns the result to the left operand	a *=b	Multiplies a and b and assign the result to 'a'. The expression could also be written as a = a*b
/=	Divides right operand by left operand and assigns the quotient to the left operand	a /=b	Divides a by b and assigns the result to a. The expression could also be written as a=a/b
%=	Divides right operand by left operand and stores remainder in the left operand	a%=b	Divides a by b and stores the remainder in a. The expression could also be written as a=a%b

Unary Operators

Operator	Description	Example	Explanation
++	Increases the value of the operand by one	a++	Equivalent to a = a+1
--	Decreases the value of the operand by one	a--	Equivalent to a = a-1

The increment operator ++, can be used in two ways- as a prefix, and as a postfix. In prefix the operator precedes the variable.

```
++var;
```

In this form the value of the variable is first incremented and then used in the expression as illustrated below:

```
var1=20;   var2 = ++var1;
```

This code is equivalent to the following set of codes:

```
var1=20;   var1 = var1+1;      var2 = var1;
```

At the end, both variables var1 and var2 store value 21.

The increment operator '++' can also be used as a postfix operator, in which the operator follows the variable.

```
var++;
```

In this case the value of the variable is used in the expression and then incremented as illustrated below:

```
var1 = 20;      var2 = var1++;
```

The equivalent of this code is:

```
var1 = 20;      var2=var1;      var1 = var1 + 1;
```

In this case, variable var1 has the value 21 while var2 remains set to 20.

In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms. If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the value of the expression is modified after the assignment.

Comparison Operators

Comparison operators evaluate to true or false. They are also called relational operators.

Operator	Description	Example	Explanation
==	Evaluates whether the operands are equal	a==b	Returns true if the operands are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=b	Returns true if the operands are not equal and false otherwise
>	Evaluates whether the left operand is greater than right operand	a>b	Returns true if the left operand is grater and false otherwise.
<	Evaluates whether the left operand is Smaller than right operand	a<b	Returns true if the left operand is smaller and false otherwise.
>=	Evaluates whether the left operand is greater or equal to the right operand	a>=b	Returns true if the left operand is equal or greater than the right operand and false otherwise.
<=	Evaluates whether the left operand is less or equal to the right operand	a<=b	Returns true if the left operand is equal or less than the right operand and false otherwise.

The increment operator ++, can be used in two ways- as a prefix, and as a postfix. In prefix the operator precedes the variable.

```
++var;
```

In this form the value of the variable is first incremented and then used in the expression as illustrated below:

```
var1=20;  var2 = ++var1;
```

This code is equivalent to the following set of codes:

```
var1=20;  var1 = var1+1;  var2 = var1;
```

At the end, both variables var1 and var2 store value 21.

The increment operator '++' can also be used as a postfix operator, in which the operator follows the variable.

```
var++;
```

In this case the value of the variable is used in the expression and then incremented as illustrated below:

```
var1 = 20;  var2 = var1++;
```

The equivalent of this code is:

```
var1 = 20;  var2=var1;  var1 = var1 + 1;
```

In this case, variable var1 has the value 21 while var2 remains set to 20.

In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms. If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the value of the expression is modified after the assignment.

Comparison Operators

Comparison operators evaluate to true or false. They are also called relational operators.

Operator	Description	Example	Explanation
==	Evaluates whether the operands are equal	a==b	Returns true if the operands are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=b	Returns true if the operands are not equal and false otherwise
>	Evaluates whether the left operand is greater than right operand	a>b	Returns true if the left operand is grater and false otherwise.
<	Evaluates whether the left operand is Smaller than right operand	a<b	Returns true if the left operand is smaller and false otherwise.
>=	Evaluates whether the left operand is greater or equal to the right operand	a>=b	Returns true if the left operand is equal or greater than the right operand and false otherwise.
<=	Evaluates whether the left operand is less or equal to the right operand	a<=b	Returns true if the left operand is equal or less than the right operand and false otherwise.

Shift Operators

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, float, or double data types.

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bit at the left	a=10 >> 3	The result of this is 10 divided by 2^3
<<	Shifts bits to the left, filling zeros at the right	a=10 << 3	The result of this is 10 multiplied by 2^3

If the int data type occupies four bytes in the memory, the rightmost eight bits of the number 10 are represented in binary as "0 0 0 0 1 0 1 0"

When you do a right shift by 3(10 >> 3), the result is "0 0 0 0 0 0 0 1", which is equivalent to 1 and when you do a left shift by 3 (10 << 3), the result is "0 1 0 1 0 0 0 0", which is equivalent to 80. Shifting an integer by one position to the left multiplies the number by 2 and shifting an integer by one position to the right divides the numbers by 2. That is "10<<1" is equivalent to "10/2" and "10>>1" is equivalent to "10 * 2"

Bit-wise Operators

Bitwise operators perform operation on operand in bit by bit fashion. Bitwise operators are described below:

Operator	Description	Example	Explanation
&(AND)	Evaluates to a binary value after bitwise AND on two operand	a&b	AND results in a 1 if both bits are 1, any other operands combination results in a 0.
|(OR)	Evaluates to a binary value after bitwise OR on two operand	a|b	OR results in a 0 if both bits are 0, any other operands combination results in a 1.
^(XOR)	Evaluates to a binary value after bitwise XOR on two operand	a^b	AND results in a 1 if both bits are same and 0 if both bits have different value.
~(inversion)	Converts all 1's into 0's and vice versa	~a	All the 1's in the byte are converted to 0s and vice versa.

Logical Operators

Logical operators are used to combine the results of two or more Boolean expressions and it again gives a Boolean value as result.

Operator	Description	Example	Explanation
&& (Logical AND)	Evaluates to true, if both the conditions are evaluated to true, false otherwise	a>6&&y<20	The result is true if condition 1 (a>6) and condition 2 (y<20) are both true. If at least one of them is false, the result is false.
|| (Logical OR)	Evaluates to false, if both the conditions are evaluated to false, true otherwise	a>6||y<20	The result is true if either condition1 (a>6) and condition 2 (y<20) or both are true. If both of them is false, the result is false.

! (Logical NOT)	Inverts the Boolean value of the operand	!(a>6)	The result is true if the condition (a>6) is false and true if the condition is false

The operators (&&, ||) appear to be similar to the bit-wise & and | operators. The two have different uses. Although in many cases (when dealing with booleans) it may appear that they have the same effect, it is important to note that the logical-OR is short circuit, meaning that if its first argument evaluates to true, then the second argument is left unevaluated. The bitwise operator evaluates both of its arguments regardless. Similarly, the logical-AND is short-circuit, meaning that if its first argument evaluates to false, then the second is left unevaluated. Again, the bitwise-AND is not. This is not always necessary since: 'false & a' would always result in false and 'true | a' would always result in true.

For example

```
a < 6 && y > 20
```

The second condition (b>20) is skipped if the first condition is false, since the entire expression will anyway, be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result, will anyway, be true. These operators, && and ||, are therefore, called short circuit operators. You can see this in action here:

```
int x = 0;
int y = 1;
int z= (x+y == 1 || y/x == 1);
int r=(x+y == 1 |  y/x == 1);
```

The first statement works just fine and returns true since the first argument evaluates to true, and hence evaluation stops. The second statement is error since it is not short circuit, and a division by zero is encountered.

Conditional Operators

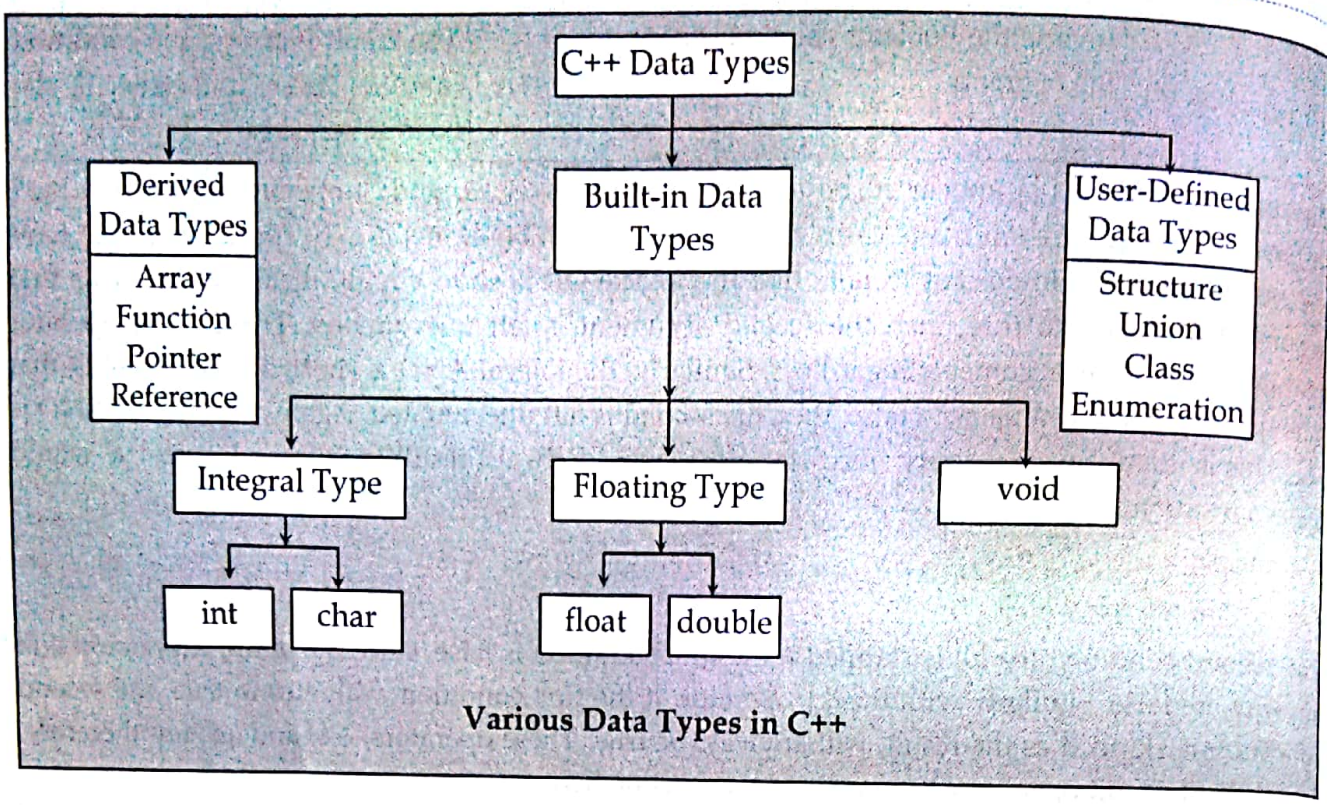
It is also called ternary operator. General form of this operator is "condition?value1:value2". If the condition evaluates to true result will be value1 and if the condition evaluates to false result will be value2. Consider the following example that calculates grade of a student based on his marks.

```
result=(marks>50?"Pass":"Fail")
```

In the above example, if marks obtained by the student is more than 50, result will b "Pass" and false otherwise.

2.3 C++ DATA TYPES

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable. It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory. The various data types provided by C++ are: *Basic Data Types, Derived Data Types and User-Defined Data Types.*



Various Data Types in C++

Basic Data Types

C++ offers the programmer a rich assortment of basic (built-in) data types. Following table lists down seven basic C++ data types.

Type	Keyword
Boolean	Bool
Character	Char
Integer Floating point	int float
Double floating point	Double
Valueless	Void

Several of the basic types can be modified using one or more of these type modifiers: *signed, unsigned, short, long*. The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
Int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647

short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

Derived Data Types

Data types that are derived from the built-in data types are known as derived data types. The various derived data types provided by C++ are *arrays, references and pointers.*

User-Defined Data Types

Various user-defined data types provided by C++ are *structures, unions, enumerations and classes.*

2.4 STRUCTURE OF A C++ PROGRAM

EXAMPLE

```
//My first program in C++
#include<iostream.h>
#include<conio.h>
int main ()
{
        cout << "Hello World!";
        getch();
        return 0;
}
```

Output

Hello World!

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
// my first program in C++
```

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

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

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive

#include <iostream.h> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

```
int main ()
```

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independent of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program. cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

```
return 0;
```

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
    cout << " Hello World!";
    getch();
    return 0;
}
```

We could have written:

```
int main () { cout << "Hello World!"; getch();return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code. In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
    cout <<
    "Hello World!";
```

```
    getch()

    return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

2.5 INPUT AND OUTPUT IN C++

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters and that these characters are provided/accepted sequentially (i.e., one after another). The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

Stream	Description
cin	Standard input stream (keyboard)
cout	Standard output stream (monitor)
cerr	Standard error stream (monitor)
clog	Standard logging stream (monitor)

2.5.1 Output with cout

cout (pronounced as 'C out') is a predefined stream object that represents the standard output stream (i.e monitor) in C++. A stream is an abstraction that refers to a flow of data. The insertion operator inserts the data that follows it into the stream that precedes it. For example, in the statement cout<<"Welcome:"; the << operator directs the string constant "Welcome" to cout, which sends it for the display to the monitor.

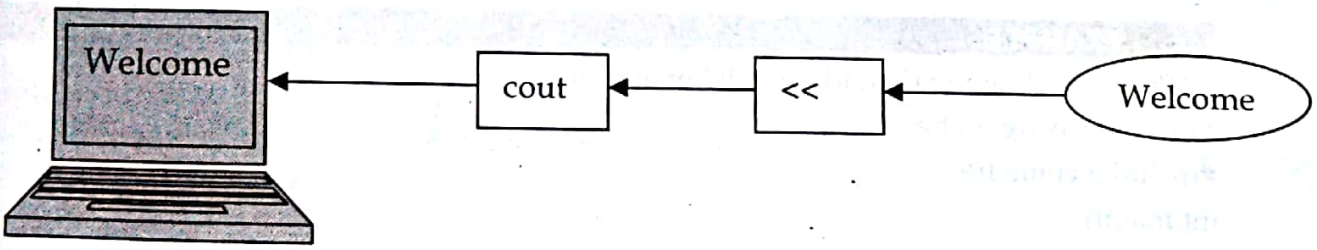


Figure 2.1: Output Using cout

2.5.2 Input with cin

cin (pronounced as 'C in') is also a stream object, predefined in C++ to correspond to the standard input stream (i.e keybord). This stream represents data coming from the keyboard. The operator >> is known as *extraction* or *get from* operator and extracts (takes) the value from the stream object cin in its left and place to the variable on its right. For example, in the statement cin>>first;, the >> operator extracts the value from cin object that is entered from the keyboard and assigns it to the variable first.

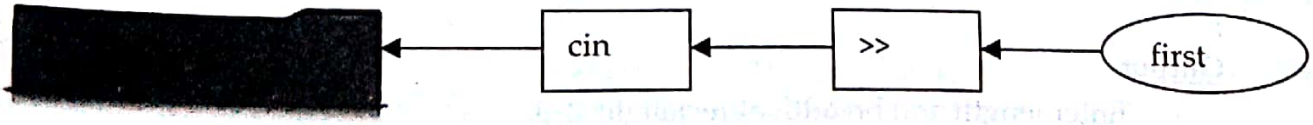


Figure 2.2: Output Using cin

EXAMPLE

```
//Using cout and cin
#include<iostream.h>
#include<conio.h>
int main()
{
        int a, l;
        cout<<"Enter length of square:\n";
        cin>>l;
        a= l * l;
        cout<<"Area=";
        cout<<a;
        getch();
        return 0;
}
```

Output

```
Enter Length of square: 3
Area=9
```

2.5.3 Cascading of I/O operators

We can use insertion operator (<<) in a cout statement repeatedly to direct a series of output streams to the cout object. The streams are directed from left to right. For example, in the statement cout<<"Sum="<<x + y;, the string "Sum=" will be directed first and then the value of x + y.

Similarly, we can use extraction operator (>>) in a cin statement repeatedly to extract a series of input streams from the keyboard and to assign these streams to the variables, allowing the user to enter a series of values. The values are assigned from left to right. For example, in the statement cin>>x>>y;, the first value entered will be assigned to the variable x and the second value entered will be assigned to variable y.

EXAMPLE

```
//Program Showing Cascading of IO operations
#include<iostream.h>
#include<conio.h>
int main()
{
        int a, l,b;
        cout<<"Enter length and breadth of rectangle:\n";
        cin>>l>>b;
        a= l * b;
        cout<<"Area="<<a;
        getch();
        return 0;
}
```

Output

```
Enter length and breadth of rectangle: 2 3
Area=6
```

2.6 THE PUT() AND GET() FUNCTIONS

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character. Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get(c); //get a character from keyboard and assign it to c
while (c!= '\n')
{
        cout << c; //display the character on screen
        cin.get (c);//get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator> >can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement 'cin >> c' is used in place of cin.get(c).

Try using both of them and compare the results. The get(void) version is used as follows:

```
char c;
c = cin.get(); //cin.get(c) replaced
```

The value returned by the function get() is assigned to the variable c. The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

```
cout << put('x'); //displays the character x and
cout << put(ch); //displays the value of variable ch.
```

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

```
cout << put(68);
```

displays the character D. This statement will convert the int value 68 to a char value and display the character whose ASCII value is 68. The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get (c); //read a character
while(c!= '\n')
{
        cout << put(c); //display the character on screen
        cin.get (c );
}
```

2:7 THE GETLINE () FUNCTION

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

```
cin.getline(line, size);
```

This function call invokes the function which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size number of characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character. For example; consider the following code:

```
char name[20];
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

'Neeraj good'

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

'Object Oriented Programming'

In this case, the input will be terminated after reading the following 19 characters: 'Object Oriented Pro'. After reading the string, cin automatically adds the terminating null character to the character array. Remember, the two blank spaces contained in the string are also taken into account, i.e. between Objects and Oriented and Pro. We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as "Neeraj good". Reading terminates either after (n-1) characters have been extracted or the delimiting character is found (which is delim if this parameter is specified, or '\n' otherwise). The extraction also stops if the end of file is reached in the input sequence or if an error occurs during the input operation.

EXAMPLE

```
// cin with strings
#include <iostream>
#include <string>

int main ()
{
 string mystr;
 cout << "What's your name? ";
 getline (cin, mystr);
 cout << "Hello " << mystr << ".\n";
 cout << "What is your favorite team? ";
 getline (cin, mystr);
 cout << "I like " << mystr << " too!\n";
 return 0;
}
```

Output

```
What's your name? Abin Saud
Hello Abin Saud
What is your favorite team? The Barcelona
I like The Barcelona too!
```

2.8 MANIPULATORS

Manipulators are the operators used with the insertion operator (<<) to modify or manipulate the way data is displayed. The most commonly used manipulators are **endl**, **setw**, and **setprecision**. Manipulators are defined in the header file "*iomanip.h*" therefore we need to include it before using manipulators. Manipulator "endl" is also defined in iostream.h therefore if we want to use only "endl", we do not need to include "iomanip.h".

2.8.1 The endl Manipulator

This manipulator causes a linefeed to be inserted into the output stream. It has the same effect as using the newline character '\n'. for example, the statement

```
cout<<"First value ="<<first<<endl<<"Second value ="<<second;
```

will cause two lines of output.

"endl" is different from newline character '\n' in the sense that it causes output buffer to be cleared after the content is displayed in the monitor but '\n' lefts the output buffer uncleared due to which we need to clear the buffer by using other instructions.

2.8.2 The setw Manipulator

This manipulator causes the output stream that follows it to be printed within a field of n characters wide, where n is the argument to setw. The output is right justified within the field. If we do not use setw, the output is left justified by default and it occupies the space in the monitor equal to number of characters in it. For example,

```
cout<<"Beautiful"<<endl<<" Kantipur"<<endl<<" Valley";
```

Output:

```
Beautiful
Kantipur
Valley
```

Effect of using setw

```
cout<<setw(11)<<"Beautiful"<<endl <<setw(11)<<"Kantipur"<<endl
<<setw(11)<<"Valley";
```

Output:

```
   Beautiful
   Kantipur
      Valley
```

2.8.3 The setprecision Manipulator

This manipulator sets the n digits of precision to the right of the decimal point to the floating point output, where n is the argument to setprecision(n). For example,

```
float a = 42.3658945, b = 35.24569, c = 58.3214789, d = 49.321489;
cout<<a<<endl <<setprecision(3)<<b<<endl<<c<<endl<<setprecision(2)<<d;
```

Output:

```
42.365894
35.246
58.321
49.32
```

The header file for setw and setprecision manipulators is **iomanip.h**.

EXAMPLE

//Program to read marks of 5 subjects and display them such that all subject names should be displayed left justified and marks should be displayed right justified. Also Display total and percentage up to the precision 2.

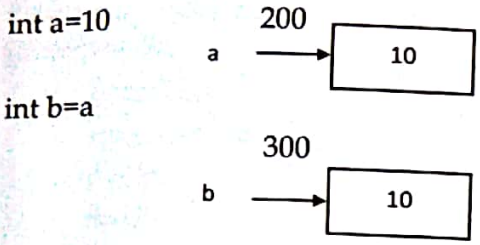
```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#include<string.h>
int main()
{
        int i,l;
        int m[5];
        char sub[5][20]={"Mathmatics","English","C Prog",
                                "Statistics","Sociology"};
        float total=0.0,per;
        cout<<"Enter marks of Mathematics, English, C Prog, Statistics
        and Sociology"<<endl;
        for(i=0;i<5;i++)
        {
            cin>>m[i];
        }
        cout<<"\t"<<"Subject"<<setw(15)<<"Marks"<<endl;
        cout<<"\t"<<"----------"<<endl;
        for(i=0;i<5;i++)
        {
                total=total+m[i];
                l=strlen(sub[i]);
                cout<<"\t"<<sub[i]<<setw(22-l)<<m[i]<<endl;
        }
        per=total/5;
        cout<<"\t"<<"----------"<<endl;
        cout<<"\t"<<"Total"<<setw(17)<<total<<endl<<"\t";
        cout<<"Percentage"<<setw(12)<<setprecision(2)<<per<<endl;
        getch();
        return 0;
}
```

Output

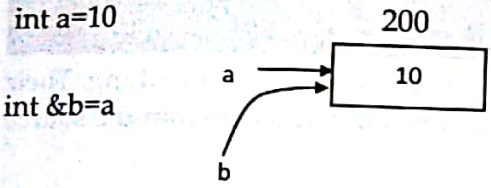
Subject	Marks
Mathematics	67
English	58
C Prog	89
Statistics	67
Sociology	56
Total	337
Persentage	67.40

2.9 REFERENCE VARIABLE

Reference variable is an alias (another name) given to the already existing variables or constants. When we declare a reference variable memory is not located for it rather it points to the memory of another variable. Reference variables are declared by preceding the variable name with & symbol.
Consider the case of normal variables



Here variable "b" points to separate memory location than variable "a" but it's value is same as value of variable "a". If we change the value of "a" value of 'b" remain unchanged and vice versa.
Now, consider the case of reference variables



Here separate memory location is not allocated for the variable b rather it points to the memory location that is allocated previously for variable "a". Since "a" and "b" both are different names provided to the same memory location, if we change the value of "a", value of b is also changed and vice versa. Program written below illustrates the concept of reference variable.

EXAMPLE

```
//Use of reference variable
#include<iostream.h>
#include<conio.h>
int main()
{
        int a=2;
        int b=a;
        cout<<"Normal Variable"<<endl;
        cout<<"a="<<a<<"\t"<<"b="<<b<<endl;
        b=b+5;
        cout<<"a="<<a<<"\t"<<"b="<<b<<endl;
/*When "b" is changed value of "a" remain unchanged because "a" and "b" points to two
different memory locations. */
//Experiment with reference variables
        int x=3;
        int &y=x;
        cout<<"Reference Variable"<<endl;
        cout<<"x="<<x<<"\t"<<"y="<<y<<endl;
```

```
        y=y+5;
        cout<<"x="<<x<<"\t"<<"y="<<y<<endl;
/*When "y" is changed value of "x" remain unchanged because "x" and "y" points to same
memory location. */
        getch();
        return 0;
}
```

Output

```
Normal Variable
a=2     b=2
a=2     b=7
Reference Variable
x=3     y=3
x=8     y=8
```

2.10 COMMENTS

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. C++ supports two ways to insert comments:

```
// line comment
/* block comment */
```

The first of them is known as line comment. It discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one is known as block comment. It discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

EXAMPLE

```
/*Program in C++ with more comments. It is better to use comments while writing the
programs because it increases understandability of the program */
#include <iostream.h>
#include<conio.h>
int main ()
{
        cout << "Hello World! "; // prints Hello World!
        cout << "I'm a C++ program"; // prints I'm a C++ program
        getch();
        return 0;
}
```

Output:

```
I'm a C++ program
```

If you include comments within the source code of your programs without using the comment characters combinations //, / and */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.*

2.11 TYPE CONVERSION

Converting an expression of a given type into another type is known as type-casting. There are two types of type conversion: automatic conversion and type casting.

2.11.1Automatic Conversion (Implicit Type Conversion)

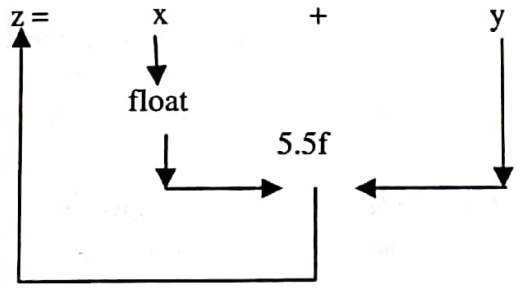
When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is called automatic type conversion or type promotion or implicit type conversion. The order of types is given below:

Data Type	Order
long double	(highest)
double	
float	
long	
int	
char	(lowest)

EXAMPLE

```
int x=3;
float y=2.5f, z;
z= x+y;
```

While above statement is compiled type promotion is performed as below:



2.11.2Type Casting

Type Casting is the forceful conversion from one type to another type. Sometimes, a programmer needs to convert a value from one type to another forcefully in a situation where the compiler will not do it automatically. For this C++ permits explicit type conversion of variables or expressions as follows:

```
(type-name) expression //C notation
type-name (expression) //C++ notation
For example,
int a = 10000;
int b = long(a) * 5 / 2; //correct
int b = a * 5/2; //incorrect
```

Here if we do not cast "a" into long, the result of sub-expression "a*5" will be 50000 which is outside of the range of data type int and causes the program to produce wrong result. But in this case automatic type promotion is not done by the compiler and hence programmer needs to use explicit type conversion to produce the right result.

EXAMPLE

```
//Program to get fractional result from division of two integers
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
int main()
{
        int x,y;
        float z;
        cout<<"Enter two numbers"<<endl;
        cin>>x>>y;
        //Since x and y both are negative, result can not be          //fractional even if z is
float
        z=x/y;
        cout<<"Result of division without          casting="<<setprecision(2)<<z<<endl;
        //To get the fractional result, any one of the operand          //involving in division
must be converted into float
        z=float(x)/y;
        cout<<"Result of division with     casting="<<setprecision(2)<<z<<endl;
        getch();
        return 0;
}
```

Output

```
Enter two numbers
10
3
Result of division without casting=3.0
Result of division with casting=3.33
```

2.12 PREPROCESSOR DIRECTIVES

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the preprocessor. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

The #define Directive

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is:

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled.

```
#include <iostream>
#define PI 3.14159
int main ()
{
  cout << "Value of PI :" << PI << endl;
  return 0;
}
```

The #include Directive

When the preprocessor finds an #include directive it replaces it by the entire content of the specified header or file. There are two ways to use #include:

```
#include <header>
#include "file"
```

In the first case, a header is specified between angle-brackets <>. This is used to include headers provided by the implementation, such as the headers that compose the standard library (iostream, string etc). The syntax used in the second #include uses quotes, and includes a file. The file is searched for in current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes ("") were replaced by angle-brackets (<>).

2.13 MEMORY MANAGEMENT WITH NEW AND DELETE

Along with malloc(), calloc() and free() functions, C++ also defines two unary operators new and delete that perform the task of allocating and freeing the memory. Since these operators manipulate memory on the free store, they are also known as free store operators. A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete.

It is a good programming practice to match every occurrence of new operator with corresponding delete operator otherwise if program runs for long time and new operator is used repeatedly without matched delete operator all memory space of heap will be exhausted and hence may cause the problem of system crash. The general form of the new operator is:

```
pointer_variable = new data_type;
For example, q = new float;
```

We can also initialize the memory using the new operator. This is done as follows:

```
pointer_variable= new data_type(value);
For example, int *p = new int(25);
```

When an object is no longer needed, it is destroyed to release the memory space for reuse. The general form is:

```
delete pointer_variable;
For example, delete p;
```

If we want to free a dynamically allocated array we must use the following form of delete.

```
delete [size] pointer_variable;
For example, delete [ ]p;
```

EXAMPLE

```
//Program to allocate and destroy the memory dynamically
#include<iostream.h>
#include<conio.h>
int main()
{
```

```
int n,i=1;
float tot_age=0,avg_age;
int a,*p;
cout<<"Enter number of students"<<endl;
cin>>n;
cout<<"Enter age of each student"<<endl;
while(i<=n)
{
        cin>>*p;
        tot_age=tot_age+a;
        i++;
        p++;
}
avg_age=tot_age/n;
cout<<"Average Age:"<<avg_age<<endl;
getch();
return 0;
}
```

Output

```
Enter number of students
5
Enter age of each student
18 18 20 19 19
Average Age:18.799999
```

The new operator has several advantages over the function malloc().

1. It automatically computes the size of the data objects.
2. It automatically returns the correct pointer type so there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Both new and delete operators can be overloaded.

The concept of arrays has a block of memory reserved. The disadvantage with the concept of arrays is that the programmer must know, while programming, the size of memory to be allocated in addition to the array size remaining constant. In programming there may be scenarios where programmers may not know the memory needed until run time. In this case, the programmer can opt to reserve as much memory as possible, assigning the maximum memory space needed to tackle this situation. This would result in wastage of unused memory spaces. Memory management operators are used to handle this situation in C++ programming language.

2.14 DECLARATION OF VARIABLES IN C++

If you are familiar with C, you would know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type it is.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes programs much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use. The following example illustrates this point.

EXAMPLE

```
//Program showing variable declaration in different places
#include<iostream.h>
#include<conio.h>
int main ( )
{
      float x; // declaration
      float sum = 0;
      for (int i = 0; i<5; i++) //declaration
      {
        cout<<"Enter number"<<endl;
        cin >> x;
        sum = sum+x;
      }
      float average; //declaration
      average = sum / 5;
      cout <<"Average="<< average;
      getch();
      return 0;
}
```

Output

```
Enter number
4
Enter number
5
Enter number
6
Enter number
7
Enter number
8
Average=6.0
```

The only disadvantage of this style of declaration is that we cannot see at a glance all the variables used in a scope.

2.15 CONST QUALIFIER

The simplest use of const qualifier is to declare a named constant with data type. This was also available in the ancestor of C++, C. To do this, we declare a constant as if it was a variable but add 'const' before it. We have to initialize it at the time of declaration.

const int PI=3.1415;

Such constants are useful for parameters which are used in the program but do not need to be changed after the program is compiled. It has an advantage for programmers over the C preprocessor '#define' command in that it is understood & used by the compiler itself, not just substituted into the program text by the preprocessor before reaching the main compiler, so error messages are much more helpful. Programs given below shows the difference between constants declared by using const modifier and constant declared by using #define directive.

EXAMPLE

```
//Program using #define to define constant
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#define PI 3,1415//compiler cannot capture error at this line
int main( )
{
        float a,r;
        cout<<"Enter radius of the circle"<<endl;
        cin>>r;
        a=PI*r*r;
        cout<<"Area of circle:"<<setprecision(4)<<a<<endl;
        getch();
        return 0;
}
```

Output

```
Enter radius of circle
2
Area of circle: 3
```

EXAMPLE

```
/*Program using const modifier to define constant, try this program and analyze the
error*/
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
int main()
{
        float a,r;
        const PI=3,1415//Compiler captures error at this line
        cout<<"Enter radius of the circle"<<endl;
        cin>>r;
        a=PI*r*r;
        cout<<"Area of circle:"<<setprecision(2)<<a<<endl;
        getch();
        return 0;
}
```

Const modifier also works with pointers but we must be careful where 'const' is put as that determines whether the pointer or what it points to is constant. For example,

```
const int *p;  or int const *p;
```

Declares that "p" is a variable pointer to a constant integer and

```
int * const p;
```

Declares that "p" is constant pointer to a variable integer and

```
int const * const p;
```

Declares that "p" is constant pointer to a constant integer. Basically 'const' applies to whatever is on its immediate left (other than if there is nothing there in which case it applies to whatever is its immediate right).

2.16 ENUMERATION

An enumeration is a user-defined type consisting of a set of named constants called enumerators. For example constants of type "int" may also be declared with an enumeration statement like below:

```
enum { SUN ,MON, TUES, WED, THURS, FRI, SAT };
```

is shorthand for

```
const int SUN = 0; const int MON = 1; const int TUES = 2; const int WED = 3;  const int
THURS = 4;  const int FRI = 5;  const int SAT = 6;
```

By default, members of an "enum" list are given the values 0, 1, 2, etc., but when "enum" members are explicitly initialized, uninitialized members of the list have values that are one more than the previous value on the list:

```
enum { SUN=1,MON, TUES, WED, THURS, FRI, SAT };
```

In this case, the value of "SUN" is 1 , and the value of "SAT" is 7.

EXAMPLE

```
//Program illustrating the use of enumeration
#include <iostream.h>
#include<conio.h>
enum Days {Sunday=1,  Monday, Tuesday, Wednesday, Thursday, Friday,Saturday};
int main(.)
{
        enum Days today = Sunday;
        switch (today)
        {
        Case 2:
        cout << "It's Monday" << endl;
        break;
        default:
        cout << "Not Monday" << endl;
        }
getch();
        return 0;
}
```

Output

```
Not Monday
```

2.17 SCOPE RESOLUTION O

The scope resolution (::) operator is used to qualify hidden names so that you can still use them. Y can use the unary scope operator if a namespace scope or global scope name is hidden by an expli declaration of the same name in a block or class. For example;

EXAMPLE 14

```
//Showing use of scope resolution operator
#include<iostream.h>
#include<conio.h>
int count = 0;
int main()
{
        int count = 0;
        ::count = 1; // set global count to 1
        cout<<"Global count="<<::count<<endl;
        count = 2; // set local count to 2
        cout<<"Local count="<<count<<endl;
        getch();
        return 0;
}
```

Output

```
Global count=1
Local count=2
```

The declaration of count declared in the main() function hides the integer named count declared i global namespace scope. The statement "::count = 1" accesses the variable named count declared i global namespace scope. Other uses of scope resolution operators will be explained in followin chapters.

2.18 NAMESPACES

Namespaces allow to group entities under a name. This way the global scope can be divided in "sub scopes", each one with its own name. The format of namespaces is:

```
namespace identifier
{
        entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that ar included within the namespace. For example:

```
namespace myNamespace
{
        int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace calle myNamespace. In order to access these variables from outside the myNamespace namespace w have to use the scope resolution operator (::). For example, to access the previous variables fro outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

EXAMPLE 15

```
//Using namespaces
#include<iostream.h>
#include<conio.h>
#include<string.h>
namespace digit
{
        int count=0;
};
namespace character
{
        int count=0;
};
namespace other
{
        int count=0;
};
int main()
{
        char str[]="1.Nepal 2.India 3.China";
        int i=0;
        while(str[i]!='\0')
        {
                if(isdigit(str[i]))
                digit::count++;
                else if(isalpha(str[i]))
                character::count++;
                else
                other::count++;
                i++;
        }
        cout<<"Number of Digits="<<digit::count<<endl;
        cout<<"Number of Characters="<<character::count<<endl;
        cout<<"Number of other Characters="<<other::count<<endl;
        getch();
        return 0;
}
```

Output

```
Number of Digits=3
Number of Characters=15
Number of other Characters=5
```

2.19 THE TYPEDEF KEYWORD

C++ provides a typedef feature that allows to define new data type names for existing data types that may be built-in, derived or user-defined data types. Once the new name has been defined, variables can be declared using this new name. For example, consider this declaration.

typedef int integer;

In this declaration, a new name integer is given to the data type into This new name now can be used to declare integer variables as shown here.

integer i, j, k;

Note that the typedef is used in a program to contribute to the development of a clearer program. Moreover, it also helps in making machine-dependent programs more portable.

Self Assessment

Fill in the Blanks.

1. Three keywords not found in ANSI C are ..... .
2. A major difference between C and C++ is the limit on the length of a.....
3. Assignment expression assigns the value of an expression, a constant or a variable to an.....
4. The data items that operators act upon are called.....
5. A declared keyword cannot be used as a ..... name.
6. The operands acted upon by arithmetic operators must represent ..... values.
7. An operation between a float and double will result in a..... .
8. A ..... can have a value of one or zero.
9. The scope of a variable extends from the point of its declaration till the end of the....., containing the declaration.
10. A reference variable provides an alias (alternative name) for a previously defined.....
11. ..... divides global scope into sub-scopes.
12. Operator that is used to access the hidden names from given block is called ..... operator.
13. ..... Is used to define user defined type with name constants.
14. Two memory management operators introduced by c++ are .....

Exercise

1. How do you differentiate variables and identifiers? Explain.
2. Does C++ provide any ternary operator? If yes, explain its working.
3. Write a C++ program to display and add five numbers.
4. Is there any advantage of const qualifier over constants defined by using #define directive? If yes, justify with suitable example.
5. What is the difference between dynamic and reference declaration of variables?
6. What is manipulator? Write a program to demonstrate the use of manipulators.
7. What are operators? What are their functions? Give examples of some unary and binary operators.
8. What are keywords? Can keywords be used as identifiers?
9. What is type casting? Explain the use of type casting with suitable example.
10. Compare and contrast the use of malloc() and free() with new and delete operator.

❑❑❑

Chapter 3

Control Structures

Objectives

After studying this unit, you will be able to:

- Identify sequential structures.
- Use selective structures.
- Understand use of looping structures.
- Apply different unconditional jump statements.

3.1 INTRODUCTION

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done to perform our program. With the introduction of control sequences we are going to have to introduce a new concept: the block of instructions. A block of instructions is a group of instructions separated by semicolons (;) but grouped in a block delimited by curly bracket signs: { and }. If we want the statement to be a single instruction we do not need to enclose it between curly-brackets ({}). If we want the statement to be more than a single instruction we must enclose them between curly brackets ({}) forming a block of instructions. Like C, C++ also supports the following three control structures:

1. Sequential structure (straight line)
2. Selection structure (branching or decision)
3. Looping structure (iteration or repetition)

3.2 SEQUENTIAL STRUCTURE

In this structure, the sequence statements are executed one after another from top to bottom in the order in which they are written.

```
statement 1;
statement 2;
statement 3;
.....
statement n;
```

In this case statement 1 is executed before statement 2, statement 2 is executed before statement 3, and so on.

3.3 SELECTION STRUCTURE

This structure makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression. The two structures of this type are: **if** and **switch**.

3.3.1 The if statements

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

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

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

For example, the following code fragment prints out x is 100 only if the value stored in variable x is indeed 100:

```
if (x == 100)
cout << "x is 100";
```

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

```
if (x == 100)
{
    cout << "x is ";
```

```
    cout << x;
}
```

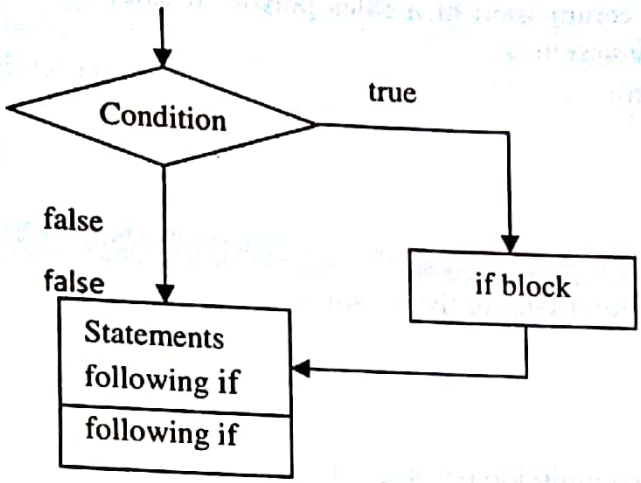


Figure 3.1: Flowchart of if statement

3.3.2 The if...else Statement

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

```
if (condition)
        statement1
else
        statement2
```

For example:

```
if (x == 100)
cout << "x is 100";
else
cout << "x is not 100";
```

Prints out on the screen *x is 100* if indeed x is worth 100, but if it is not -and only if not- it prints out *x is not 100*.

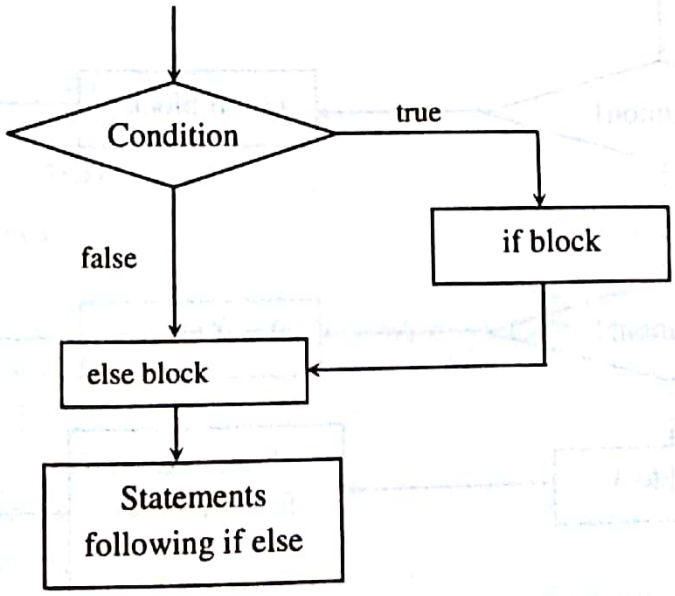


Figure 3.2: Flowchart of if else statement

EXAMPLE

```
/*Program to find commission of a sales person, if sales is more than 5000 then sales person gets 5% commission.*/
#include<iostream.h>
#include<conio.h>
int main()
{
        int s,c;
        cout<<"Enter total sales of the person:";
        cin>>s;
        c=(5*s)/100;
        if(s>=5000)
        {     cout<<"commission is="<<c;   }
        else
        {     cout<<"commissionis zero";  }
        getch();
}
```

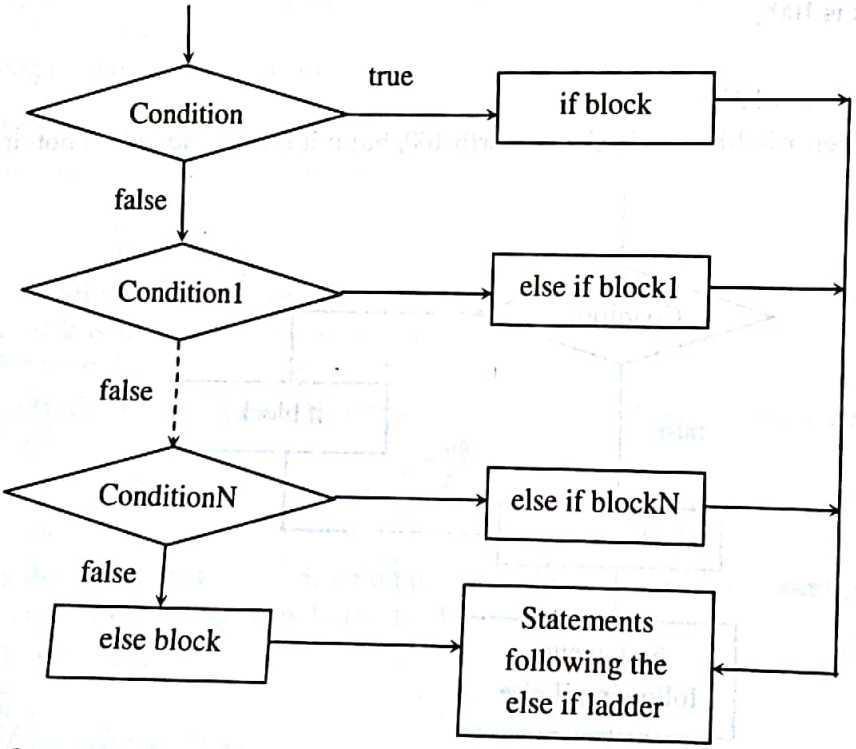
Output

```
Enter total sales of the person: 6000
Commission is=300
```

Remember that we do not need to enclose instruction blocks into curly brackets if the block contains only one instruction.

3.3.3 The else...if Ladder

The **if + else** structures can be concatenated with the intention of verifying a range of values.



ure 3.3: Flowchart of else if ladder

The following example shows its use telling if the present value stored in x is positive, negative or none of the previous, that is to say, equal to zero.

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

EXAMPLE

```
//Program to calculate tax and net salary from gross salary.
#include<iostream.h>
#include <conio.h>
int main()
{
        float gs,ns,tax;
        cout<<"Enter Gross Salary"<<endl;
        cin>>gs;
        if(gs>40000)
        tax=gs*20/100;
        else if(gs>30000)
        tax=gs*15/100;
        else if(gs>20000)
        tax=gs*10/100;
        else
        tax=0;
        ns=gs-tax;
        cout<<"Tax="<<tax<<endl;
        cout<<"Net Salary="<<ns<<endl;
        getch();
}
```

Output

```
Enter Gross Salary: 25000
Tax=2500
Net Salary=22500
```

3.3.4 The switch statement

It is also called multi way decision statement. If the use of multiple statements needed in our program then in this case we can also use the switch statement. Use of this decreases the complexity of the program. The syntax is of the form:

```
switch (expression)
{
        case value 1:
                statement(s);
```

```
            break;
        case value 2:
            statement(s)
            break;
            .....
        case value n:
            statement(s);
            break;
            default:
            statement(s);
    }
```

In this case, the value of the expression is compared with each of the constant values in the case statements. If a match is found, the program executes the statement(s) following the matched case statement. If none of the constants matches the value of the expression, then the program executes the statement(s) following default statement. However, the default statement is optional.

The value of the expression must be of type integer or character and each of the values specified in the case statement must be of a type compatible with the expression. Each case value must be a unique literal. Duplicate case values are not allowed. The break statement is used to terminate the entire switch statement.

EXAMPLE

```
//Program to identify day of week
#include<iostream.h>
#include<conio.h>
int main()
{
    int d;
    cout<<"Enter day of week:";
    cin>>d;
    switch(d)
    {
    case1:
        cout<<"The day is sunday";
        break;
    case 2:
        cout<<"The day is monday";
        break;
    case 3:
        cout<<"The day is tuesday";
        break;
    case 4:
        cout<<"The day is wednesday";
        break;
    case 5:
```

```
            cout<<"The day is thursday";
            break;
        case 6:
            cout<<"The day is friday";
            break;
        case 7:
            cout<<"The day is saturday";
            break;
        default:
            cout<<"Sorry!!!!!!! Wrong day";
    }
        getch();
    }
```

Output

```
Enter day of week: 3
The day is Tuesday
```

If multiple case values points to the same thing we need to write multiple case values serially and then write common block of statements for all case values as below:

```
switch( month )
{
    case 4:
    case 6:
    case 9:
    case 11:
            cout<< "30 days";
            break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
            cout<< "31 days";
            break;
    case 2:
            cout<< "if leap year 29 days else 28 days";
            break;
    default:
            cout<< "Number for months must be 1-12";
 }
```

If month is 4, 6, 9, or 11, then "30 days" is printed. If month is 1, 3, 5, 7, 8, 10, or 12, then "31 days" is printed. If month is 2, then "if leap year 29 days else 28 days" is printed. Lastly, if the month value is

not between 1 and 12 no case is matched, so the default statement "Nnumber for months must be 1. 12" is printed.

3.4 LOOPING STRUCTURE

These structures repeatedly execute a section of your program a certain number of times. The repetition continues until a condition is true. When the condition becomes false, the loop ends and control passes to the statement following the loop. C++ provides three kinds of loops: the for loop, the while loop, and the do…. while loop.

3.4.1 The for loop

The general form is:

```
for(initialization; test expression; increment)
{
        Statement(s);
}
```

Before loop starts, the initialization statement is executed that initializes the loop control variable or variables in the loop. Second the test expression is evaluated, if it is true, the program executes the body of loop. Finally, the iteration (usually an expression that increments or decrements the loop control variables) statement will be executed, and the test expression will be evaluated again, this continues until the test expression is false. The curly braces are unnecessary if only a single statement is being repeated.

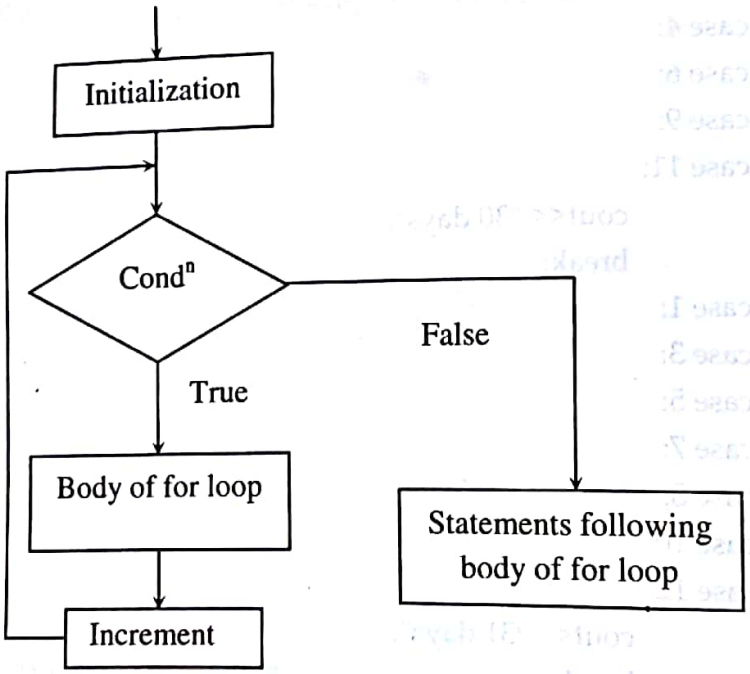


Figure 3.4: Flowchart of for loop

EXAMPLE

```
//Program to display the numbers 0 through 9
#include <iostream.h>
#include<conio.h>
int main ( )
{
```

```
    int digit;
    for (digit=0;digit<=9; ++digit)
      cout<<digit<<"\t";
    getch( );
    return 0;
}
```

Output

0 1 2 3 4 5 6 7 8 9

3.4.2 The while loop

The general form is:

```
initialization
while(test expression)
{
        statement(s);
        increment/decrement of control variable
}
```

Here the program executes the body of loop as long as the test expression is true. When the test expression becomes false, the while loop stops executing its body. The curly braces are unnecessary if only a single statement is being repeated.

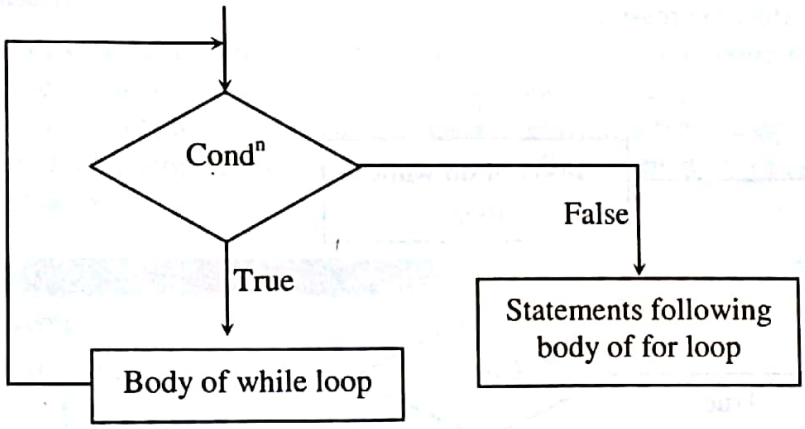


Figure 3.5: Flow Chart of while loop.

EXAMPLE

```
//Program to generates the Fibonacci series between 1 and 100.
#include<iostream.h>
#include<conio.h>
int main ( )
{
        int num1 = 1, num2 = 1;
        cout << num1 <<"\t";
        while ( num2 < 100)
        {
        cout << num2 <<"\t";
```

```
        num2 += num1;
        num1 = num2 - num1;
        }
    getch();
    return 0;
}
```

Output

1 1 2 3 5 8 13 21 34 55 89

A common error when using the while loop is to forget to do anything that might change the value being tested on while statement within the loop. If the variable being tested is not changed within the loop then the condition will either never be true and the loop will therefore never get executed or it will always be true and will execute forever.

3.4.3 The do-while loop

The general form is:

```
initialization
do
{
    statement(s);
    increment/decrement of control variable
} while(test expression);
```

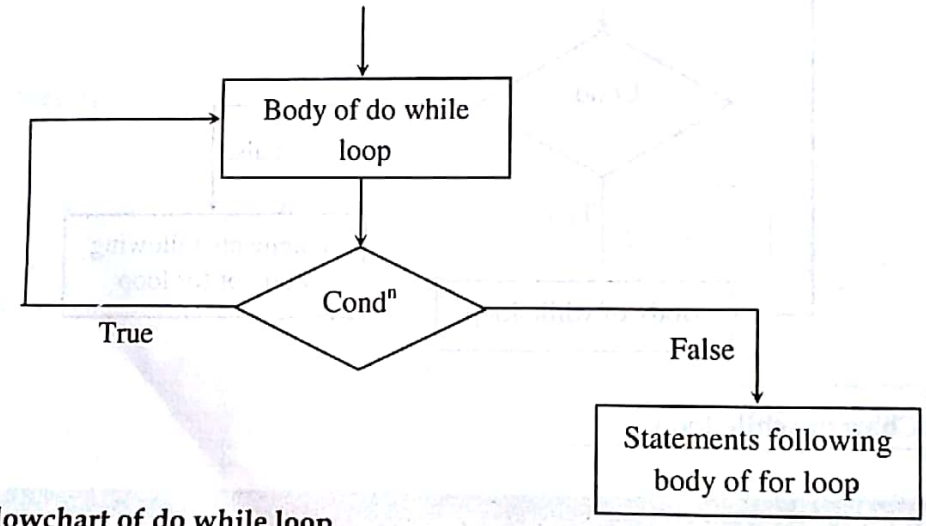


Figure 3.6: Flowchart of do while loop.

In this case, the program executes the body of loop first and then evaluates the test expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. The curly braces are unnecessary if only a single statement is being repeated. This loop always executes body of loop at least once, because its test expression is at the bottom of the loop.

EXAMPLE

```
//Program to calculate reverse of a number
#include<iostream.h>
#include<conio.h>
int main()
```

```
{
  int num,rev=0,rem;
  cout<<"Enter a number:";
  cin>>num;
  do{
              rem=num%10;   rev=rev*10+rem;
              num=num/10;
     }while(num>0);
  cout<<"Reverse="<<rev<<endl;
  getch();
  return 0;
}
```

Output

Enter a number: 672
Reverse=276

3.5 JUMP STATEMENTS

Jump statements are used to transfer control of a program execution unconditionally from one part of the program to another. C++ supports three jump statements: break, continue, and return.

3.5.1 The break Statement

The use of break statement causes the immediate termination of the switch statement and the loop (all type of loops) from the point of break statement. The control then passes to the statements following the switch statement and the loop. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before it naturally finishes (an engine failure maybe):

EXAMPLE

```
//Break loop example
#include <iostream.h>
#include<conio.h>
int main ( )
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  getch();   return 0;
}
```

Output

```
10 9 8 7 6 5 4 3 countdown aborted!
```

In the code above the loop is terminated immediately after the value of i is 3. In case of nested loop if break statement is in inner loop only the inner loop is terminated.

3.5.2 The continue Statement

The continue instruction causes the program to skip the rest of the loop in the present iteration as if the end of the statement block would have been reached, causing it to jump to the following iteration. For example, we are going to skip the number 5 in our countdown:

EXAMPLE

```
//Continue loop example
#include <iostream.h>
#include<conio.h>
int main( )
{
        for(int n = 0; n<6; n++)
    {
        cout << "n = " << n ;
        if(n%2 == 1) continue;
        cout << "This is even\n";
    }
    getch( );        return 0;
}
```

Output

```
n = 0    This is even
n = 1
n = 2    This is even
n = 3
n = 4    This is even
n = 5
```

3.5.3 The goto Statement

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:). This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

EXAMPLE

```
// goto loop example
#include <iostream.h>
#include<conio.h>
int main ()
{
```

```
        int n=10;
        loop:
        cout << n << ", ";
        n--;
        if (n>0)
                goto loop;
        cout << "FIRE!";
        getch( );
        return 0;
}
```

Output

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

3.5.4 The return Statement

It is used to transfer the program control back to the caller of the method. There are two forms of return statement. First with general form return expression; returns the value whereas the second with the form return; returns no value but only the control to the caller.

Self Assessment

Fill in the Blanks.

1. The ..... is used to carry out a logical test and then take on of two possible actions, depending on the outcome of the test.
2. The ..... statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.
3. ..... statement is used to select any one of the alternative out of many available alternatives based on value.
4. The statement which abort the current iteration of the loop and starts another iteration is called ..... statement.
5. The ..... statement causes the program flow to exit the body of the while loop.
6. ..... statement can be used to come out from the nested loop unconditionally.
7. Looping statements that evaluates the conditions at the beginning of loop are called ..... loops.
8. ..... Loop causes the body of loop to be executed at least once irrespective to the condition specified.
9. Return statement without expression transfers ..... from called function to calling function.
10. Out of three expressions of a for loop ..... is performed only once.

Exercise

1. Are 'break' and 'continue' bad programming practices? Analyze.
2. Does the last element in a loop deserve a separate treatment? Why or why not?
3. Which useful alternative control structures do you know? Explain with an example.
4. Does C++ provide any ternary operator? If yes, explain its working.

5. Convert the following while loop into for loop.

```
int i = 10;
while(i < 20)
{
        cout << i;
        (i++)++;
}
```

6. In control structure switch-case what is the purpose of default in C++?
7. Write a C++ program to display and add five numbers.
8. Explain different forms of return statement with suitable code segments.
9. Write a program that uses two nested for loops and the modulus operator (%) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1).
10. Write the program to determine the sum of the harmonic series (1 + 1/2 + 1/3 + 1/4 + ... + 1/n) for a given value of n.
11. Write a program to find the sum of the series $1 + x^2 + 3x^2 + 4x^2 + \ldots + nx^2$.
12. Write a program that calculates the sequence 1/1! + 2/2! + 3/3! +.....+ n/n! Where n is the number of input by the user.
13. Write a program to display sum of the following series up to n terms. Sum = $x - x^2 + x^3 - x^4$ +.....

❑❑❑

Chapter 4

Pointers and Functions

Objectives

After studying this unit, you will be able to:

- Recognize the functions.
- Describe the function overloading.
- Explain the inline functions.
- Discuss the default arguments.
- Describe the function prototyping.

4.1 WHAT ARE POINTERS?

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
int   *ip;  // pointer to an integer
double *dp;  // pointer to a double
float  *fp;  // pointer to a float
char   *ch   // pointer to character
```

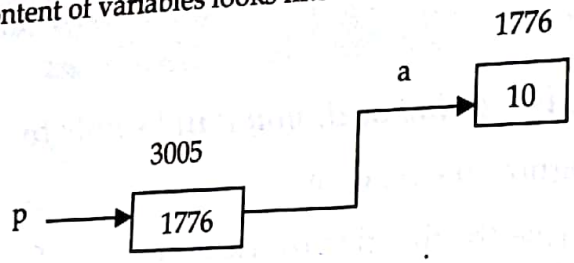
The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

4.2 ADDRESS-OF AND DEREFERENCING OPERATOR

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example:

```
int a=10;
int *p =&a ;       //pointer declaration and initialization
```

This would assign the address of variable a to pointer variable p. We are no longer assigning the content of the variable itself to p, but its address. The actual address of a variable in memory cannot be known before runtime. In order to help clarify the concepts lets assume that a is placed during runtime in the memory address 1776 and p is placed in memory address 3005. Now memory allocation and content of variables looks like below:



An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as *value pointed to by*. Consider the statement given below:

```
int x = *p;
```

This could be read as x equal to value pointed to by p, and the statement would actually assign the value 10 to x.

EXAMPLE

```
#include <iostream.h>
int main ()
{
        int  v = 20;  // actual variable declaration.
        int  *p;      // pointer variable
```

```
    p = &v;     // store address of var in pointer variable
    cout << "Value of v variable:"<<v<<endl;
    cout << "Address stored in p variable:"<<p << endl;
    cout << "Value of *p variable:"<<*p << endl;
    return 0;
}
```

Output

```
Value of v variable: 20
Address stored in p variable: 0xbfc601ac
Value of *p variable: 20
```

4.3 POINTER AND ARRAYS

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int x[20];
int * p;
```

Now, the following assignment operation would be valid:

```
p= x;
```

After that, pointer p and x would be equivalent and would have very similar properties. The main difference being that p can be assigned a different address, whereas x can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

```
x = p;   //not valid
```

In arrays, brackets ([ ]) are used for specifying the index of an element of the array. In fact these brackets are a dereferencing operator known as offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.

```
a[5] = 0;     // a [offset of 5] = 0
*(a+5) = 0;   // pointed to by (a+5) = 0
```

These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array. Remember that if an array, its name can be used just like a pointer to its first element.

EXAMPLE

```
#include <iostream.h>
const int MAX = 3;
int main ()
{
    int  var[MAX] = {10, 100, 200};
    int  *ptr;
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = "<< *ptr << endl;
        ptr++;  // point to the next location
```

```
        }
        return 0;
    }
```

Output

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

4.4 INTRODUCTION TO FUNCTIONS

Those who are familiar with C language would agree that writing a C program is nothing more than writing C functions (including the main function). C++ on the other hand is all about writing codes for defining and manipulating classes and objects. Conceptually an object may have only data members specifying its attributes. However, such an object would serve no useful purpose. For the purpose of establishing communication with the object it is necessary that the object provide methods, which are C like functions. Though C++ functions are very similar to C functions, yet they differ significantly as you will discover in this unit.

The Main Function

An application written in C++ may have a number of classes. One of these classes must contain one (and only one) method called main method. Although a private main method is permissible in C++ it is seldom used. For all practical purposes the main method should be declared as public method. The main method can take various forms as listed below:

```
main()
main(void)
void main()
void main(void)
int main()
int main(void)
int main(int argc, char *argv[])
main(int argc, char *argv[])
void main(int argc, char *argv[])
```

As is evident from the above forms, return type of the main method specifies the value returned to the operating system once the program finishes its execution. If your main method does not return any value to the operating system (caller of this method), then return type should be specified as void.

4.5 FUNCTION PROTOTYPING

Prototype of a function is the function without its body. The C++ compiler needs to about a function before you call it, and you can let the compiler know about a function in two ways:

- By defining it before you call it or
- By specifying the function prototypes before you call it

Many programmers prefer a Top-Down approach in which main() appears ahead the user-defined function definition. In such approach, function call will precede function definition which cause compiler error. To overcome this, we use the function prototypes or we declare the function. Function-prototype is usually written at the beginning of a program, ahead of any programmer-defined functions including main (). In general, function prototype is written as:

return-type name (type 1 arg. 1, type 2 arg. 2, ... type n arg. n);

Where return-type represents the data type of the value returned by the function, name represents the function name, type 1, type 2 .... type n represents the data-type of arguments arg 1, arg 2...arg n. The function prototypes resemble first line to function definition, though it ends with the semicolon. Within the function declaration, the names of arguments are optional but data-types are necessary, for example:

```
int count (int);
```

The code snippet given below will cause compilation error because the main function does not know about the called function one().

```
void main()
{
...
one(); //incorrect!! No function prototype available for one()
...
}
void one()
{
//function definition
}
```

To resolve this problem you should define the function before it is called as shown below:

```
void one()
{
//function definition
}
void main()
{
...
one(); //Correct!! Function one is defined before calling
...
}
```

Another elegant alternative to this problem is to include a function prototype before it called no matter where the function has been actually defined as shown below:

```
void one(void); //prototype of function one()
void main()
{
...
one(); //Correct!! Function one is declared before calling
...
}
void one()
{
//function definition
}
```

4.6 DEFAULT ARGUMENTS

When declaring a function we can specify a default value for each of the last parameters which are called default arguments. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

EXAMPLE

```
//Default values in functions
#include <iostream.h>
#include<conio.h>
int divide (int a, int b=2)
{
        int r;
        r=a/b;
        return (r);
}
int main ()
{
        cout << divide (12);
        cout << endl;
        cout << divide (20,4);
        getch();
        return 0;
}
```

Output

```
6
5
```

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

We have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

divide (20,4)

There are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).We can assign default values only to the trailing arguments. Assigning values to other arguments without assigning values to the trailing arguments generates a compilation error.

4.7 MACROS

Function call is a costly operation. During the function call and it's execution our system takes overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading the registers with previously stored values. For large functions this overhead is negligible but for small function taking such large overhead is not justifiable.

Macros are designed to reduce the function call overhead in case of small functions. Concept of macros are not new to c++, they are also supported by C language. Macros are defined by using hash sign because they are processed by macro-processors not by compilers. We can define and use the macros as shown in the program below:

EXAMPLE

```
//Use of macros
#include <iostream.h>
#include<conio.h>
#define mul(a,b) a*b //macro definition
#define div(a,b) a/b
int main ()
{
        int x=2,y=3;
        float n=5.0,m=2.0;
        cout << mul (x,y);// macro call
        cout << "\n";
        cout << div(n,m);//macro call
        cout << "\n";
        getch();
        return 0;
}
```

Output

```
6
2.5
```

All macro call statements are replaced by their definition by macro-processor and then after compilation of the program is performed. Therefore control needs not to be transferred to the macro definition at the time of program execution. But main drawback of macros is that macros are not checked for errors, they are substituted by macro-processor as it is.

The key to the problems of preprocessor macros is that you can be fooled into thinking that the behavior of the preprocessor is the same as the behavior of the compiler. Of course, it was intended that a macro look and act like a function call, so it's quite easy to fall into this fiction. If we use expressions as arguments to the macros, result generated by macros may be quite different from our expectation.

EXAMPLE

```
//Program showing limitation of macro
#include <iostream.h>
#include<conio.h>
```

```
#define mul(a,b) a*b
int main ()
{
  int result;
  result=mul(2+4,5);
  cout<<"Result="<<result<<endl;
  getch();
  return 0;
}
```

Output

Result=22

When macro call mul(2+4,5) is made, rather than multiplying 6 and 5 macro is expanded as "2+4*5" and thus result becomes 22.

4.8 INLINE FUNCTIONS

By combing the advantages of both functions and macros, C++ introduces the concept of inline function. A function which is expanded inline by the compiler each time its call is appeared instead of jumping to the called function as usual is called inline function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

EXAMPLE

```
//Inline Function
#include <iostream.h>
#include<conio.h>
inline void sum(int a, int b)
{
    int s;
    s= a+b;
    cout<<"Sum="<<s<<endl;
}
int main()
{
    int x, y;
    cout<<"Enter two numbers"<<endl;
    cin>>x>>y;
    sum(x,y);
    getch();
    return 0;
}
```

Output

```
Enter two numbers
4
9
Sum=13
```

Here, at the time of function call instead of jumping to the called function, function call statement is replaced by the body of the function. So there is no function call overhead.

Advantages

- No function call overhead, therefore execution of the program becomes faster than using functions
- Better error checking is performed as compared to macros
- Can access members of a class which cannot be done with macros

Drawbacks

- If the function call is made repeatedly, object code size increases and thus requires more memory at the time of execution.
- The compiler cannot perform inlining if the function is too complicated. For example, function cannot be expanded inline if it contains loops or it is recursive.
- The compiler also cannot perform inlining if the address of the function is taken implicitly or explicitly.

4.9 FUNCTION OVERLOADING

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. This is called function overloading. This can be very tiring and extremely difficult to remember all the names. Function overloading is a mechanism that allows a single function name to be used for different functions. The compiler does the rest of the job. It matches the argument numbers and types to determine which functions are being called. The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature.

EXAMPLE

```
//Function overloading with different type of arguments
#include <iostream.h>
#include<conio.h>
int mul (int a, int b)
{
        return (a*b);
}
float mul (float a, float b)
{
        return (a*b);
}
int main ()
{
        int x=5,y=2;
```

```
    float n=7.0,m=3.0;
    cout <<"Product of integers:"<< mul (x,y);
    cout << "\n";
    cout <<"Product of reals:"<< mul(n,m);
    cout << "\n";
    getch();
    return 0;
}
```

Output

```
Product of integers:10
Product of reals:10.0
```

In the first call to *mul* the two arguments passed to the function are of type int, therefore, the function with the first prototype is called. This function returns the result of multiplying both parameters. In the second call to *mul* the two arguments passed to the function are of type float, therefore, the function with the second prototype is called. Thus behavior of a call to *mul* depends on the type of the arguments passed because the function has been overloaded.

EXAMPLE

```
//Overloading Functions with different number of arguments
#include<iostream.h>
#include<conio.h>
int area(int l, int b)
{
    return l*b;
}
int area(int l)
{
    return l*l;
}
int main()
{
    int sa,ra,l,b;
    cout<<"Enter length and breadth of rectangle"<<endl;
    cin>>l>>b;
    ra=area(l,b);
    cout<<"Enter length of a side of an square"<<endl;
    cin>>l;
    sa=area(l);
    cout<<"Area of Rectangle="<<ra<<endl;
    cout<<"Area of Square="<<sa<<endl;
    getch();
    return 0;
}
```

Output

```
Enter length and brfeadth of rectangle
```

```
8    4
Enter length of a side of square
7
Area of Rectangle=32
Area of square= 49
```

A function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

4.10 PASSING ARGUMENTS TO THE FUNCTION

We can pass arguments to the function in three ways: *Pass by value, Pass by reference, and pass by pointer.*

4.10.1 Pass By Value

In case of pass by value, Copies of the arguments are passed to the function not the variables themselves. For example, suppose that we called our function exchange using the following code:

EXAMPLE

```
//Pass by value
#include <iostream.h>
#include <conio.h>
void  exchange (int a, int b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main ()
{
        int x=5, y=3;
        exchange (x,y);
        cout << "After function call " <<endl;
        cout<<"x="<<x<<endl<<"y="<<y<<endl;
        getch();
        return 0;
}
```

Output

```
After function call
x=5
y=3
```

What we did in this case was to call to function exchange passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves. When the function exchange is called, any modification to either 'a' or 'b' within the function exchange will not have any effect in the values of x and y outside it. This is because variables 'x' and 'y' were not themselves passed to the function, but only copies of their values at the moment the function was called. Main drawback of pass by value is copying large structs or classes can take a lot of time to copy, and this can cause a performance penalty, especially if the function is called many times.

4.10.2 Pass By Reference

In case of pass by reference, address of the variables (variable itself) not copies of the arguments are passed to the function. For example, suppose that we called our function exchange using the following code:

EXAMPLE

```
//Pass by Reference
#include <iostream.h>
#include <conio.h>
void  exchange (int &a, int &b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main ()
{
        int x=5, y=3;
        exchange (x,y);
        cout << "After function call " <<endl;
        cout<<"x="<<x<<endl<<"y="<<y<<endl;
        getch();
        return 0;}
```

Output

```
After function call
x=3
y=5
```

What we did in this case was to call to function exchange passing the variables x and y themselves (not values 5 and 3) respectively. When the function exchange is called, the value of its local variables 'a' and 'b' points to the same memory location as that of variables 'x' and 'y' respectively, therefore any modification to either 'a' or 'b' within the function exchange will also have effect in the values of x and y outside it.

4.10.3 Pass By Pointer

Working principle of pass by pointer is same as the pass by reference. Only the difference is that instead of using the reference variable we use the pointer variable in function definition and pass address of the variable from function call statement as below:

EXAMPLE

```
//Pass by Pointer
#include <iostream.h>
#include <conio.h>
void  exchange (int *a, int *b)
{
        int temp;
```

```
        temp=*a;
        *a=*b;
        *b=temp;
}
int main ()
{
        int x=5, y=3;
        exchange (&x,&y);
        cout << "After function call " <<endl;
        cout<<"x="<<x<<endl<<"y="<<y<<endl;
        getch();
        return 0;
}
```

Output

```
        After function call
x=3
y=5
```

4.11 CONST ARGUMENTS

When arguments are passed by reference to the function, the function can modify the variables in the calling program. If we use the constant arguments in the function, the variables in the calling program can not be modified. const qualifier is used for it.

```
void func(int&,const int&);
void main()
{
    int a=10, b=20;
    func(a,b);
}
void func(int& x, int &y)
{
    x=100;
    y=200; // error since y is constant argument
}
```

4.12 RETURNING FROM FUNCTION

Functions can return value as well as memory location. We can return values from function in three ways: *by value, by reference, and by pointer*. Returning values from a function to its caller by value, address, or reference works almost exactly the same way as passing parameters to a function does. The primary difference between the two is simply that the direction of data flow is reversed. However, there is one more added bit of complexity – because local variables in a function go out of scope when the function returns, we need to consider the effect of this on each return type.

4.12.1 Return by Value

Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller. As with pass by value, you can return by value literals (eg. 5), variables (eg. x), or expressions (eg. x+1), which makes return by value very flexible. Another advantage of return by value is that you can return variables (or expressions) that involve local

variables declared within the function. Because the variables are evaluated before the function goes out of scope, and a copy of the value is returned to the caller, there are no problems when the variable goes out of scope at the end of the function. Return by value is the most appropriate when returning variables that were declared inside the function, or for returning function arguments that were passed by value. However, like pass by value, return by value is slow for structures and large classes.

EXAMPLE

```
//Return by value
#include<iostream.h>
#include<conio.h>
float calculate_amount(float p, float t, float r)
{
        float si;
        si=(p*t*r)/100;
        return(p+si);
}
int main()
{
        float p,t,r,a;
        cout<<"Enter values of p, t, r"<<endl;
        cin>>p>>t>>r;
        a=calculate_amount(p,t,r);
        cout<<"amount="<<a<<endl;
        getch();
        return 0;
}
```

Output

```
Enter values of p, t, r
2000
3
14
Amount=2840
```

4.12.2 Return by Reference

Just like with pass by reference, values returned by reference must be variables (you can not return a reference to a literal or an expression). When a variable is returned by reference, a reference to the variable is passed back to the caller. Return by reference is also fast, which can be useful when returning structures and classes. However, returning by reference has one additional downside that pass by reference doesn't – you can't return local variables to the function by reference. If we try this, the function will try to return a reference to a value that is going to go out of scope when the function returns. This would mean the caller receives a reference to garbage. Fortunately, your compiler will give you an error if you try to do this. Return by reference is typically used to return arguments passed by reference to the function back to the caller.

EXAMPLE

```
//Return by reference
#include<iostream.h>
```

```
#include<conio.h>
int& min(int &x, int &y)
{
        if(x<y)
        return x;
        else
        return y;
}
int main()
{
        int a,b,r;
        cout<<"Enter two numbers"<<endl;
        cin>>a>>b;
         min(a,b)=0;
        cout<<"a="<<a<<endl<<"b="<<b;
        getch();
        return 0;
}
```

Output

First execution:

```
Enter two numbers
8
3
a=8
b=0
```

Second execution:

```
Enter two numbers
5
19
a=0
b=19
```

Here min function return the reference of the variable which is smaller between a and b and the statement min(a, b)=0 makes the value of the variable zero which is smaller.

4.12.3 Return by Pointer

Returning by pointer involves returning the address of a variable to the caller. Just like pass by pointer, return by pointer can only return the address of a variable, not a literal or an expression. Like return by reference, return by address is fast. However, as with return by reference, return by address can not return local variables:

EXAMPLE

```
//Return by pointer
#include<iostream.h>
#include<conio.h>
int* min(int *x, int *y)
{
        if(*x<*y)
```

```
        return x;
        else
        return y;
}
int main()
{
        int a,b;
        cout<<"Enter two numbers"<<endl;
        cin>>a>>b;
        int *m=min(&a,&b);
        cout<<"Smaller Element is:"<<*m<<endl;
        getch();
        return 0;
}
```

Output

First execution:

```
Enter two numbers
8
3
Smaller Element is:3
```

Second execution:

```
Enter two numbers
5
19
Smaller Element is:5
```

Here min function return the address of the variable which is smaller between a and b and the statement min(a, b)=0 makes the value of the variable zero which is smaller.

4.13 SCOPE AND STORAGE CLASSES IN C++

4.13.1 Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces, in which variable is declared that a variable exists, not outside it. There are two types of variable scope:

- Local Scope
- Global Scope (File Scope)
- Class Scope

Local Scope

A variable declared within a block of code enclosed by braces ({ }) is accessible only within that block, and only after the point of declaration. Outside that they are unavailable and leads to compile time error.

Global Scope

Any variable declared outside all blocks or classes has global scope. It is accessible anywhere in the file after its declaration.

Class Scope

Members of classes have class scope. Variables with class scope are accessible in all of the methods of the class. Details of class scope will be considered in upcoming chapters.

EXAMPLE

```
#include <iostream.h>
int g=20; //global variable
int main()
{
        int i=10; //local variable
        if(i<20)      // if condition scope starts
        {
                int n=100;  // Local variable
                cout<<"Inside if"<<endl;
                cout<<"g="<<g<<endl;
                cout<<"i="<<i<<endl;
                cout<<"n="<<n<<endl;
        } // if condition scope ends
        cout<<"Below if"<<endl;
        cout<<"g="<<g<<endl;
        cout<<"i="<<i<<endl;
        //cout<<"n="<<n<<endl; ----error cannot be accessed here
}
```

Output

```
Inside if
g=20
i=10
n=100
Below if
g=20
i=10
```

4.13.2 Storage Classes

Storage class of a variable also defines the lifetime and scope (visibility) of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible. There are five types of storage classes in C++. They are:

- Automatic
- External
- Static
- Register
- Mutable

Automatic Storage Class

The Keyword *auto* is used to declare automatic variables. However, if a variable is declared without any keyword inside a function, it is automatic by default. This variable is visible only within the function it is declared and its lifetime is same as the lifetime of the function as well. Once the execution of function is finished, the variable is destroyed.

External Storage Class

External storage class assigns variable a reference to a global variable declared outside the given program. The keyword *extern* is used to declare external variables. They are visible throughout the program and its lifetime is same as the lifetime of the program where it is declared. This visible to all the functions present in the program.

Static Storage Class

Static storage class ensures a variable has the visibility mode of a local variable but lifetime of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished. When a function is called, the variable defined as static inside the function retains its previous value and operates on it. This is mostly used to save values in a recursive function.

Register Storage Class

Register storage assigns a variable's storage in the CPU registers rather than primary memory. It has its lifetime and visibility same as automatic variable. The purpose of creating register variable is to increase access speed and makes program run faster. If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class. So only those variables which requires fast access should be made register. Keyword register is used for specifying register variables.

Mutable Storage Class

The mutable specifier applies only to objects. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function. We will discuss about it in detail in upcoming chapters.

EXAMPLE

```
#include<iostream.h>
int g;   //global variable, initially holds 0
void test()
{
  static int s;   //static variable, initially holds 0
  register int r;   //register variable
  r=5;
  s=s+r*2;
  cout<<"Inside test"<<endl;
  cout<<"g = "<<g<<endl;
  cout<<"s = "<<s<<endl;
  cout<<"r = "<<r<<endl;
}
int main()
{
  int a;   //or declare as: auto int a;          automatic variable
  g=25;
  a=17;
  test();
  cout<<"Inside main"<<endl;
  cout<<"a = "<<a<<endl;
  cout<<"g = "<<g<<endl;
  test();
  return 0;
}
```

Output

```
Inside test
g = 25
s = 10
r = 5
Inside main
a = 17
g = 25
Inside test
g = 25
s = 20
r = 5
```

In the above program, g is a global variable, s is static, r is register and a is automatic variable. We have defined two function, first is main() and another is test(). Since g is global variable, it can be used in both function. Variables r and s are declared inside test() so can only be used inside that function. However, s being static isn't destroyed until the program ends. When test() is called for the first time, r is initialized to 5 and the value of s is 10. After the termination of test(), r is destroyed but s still holds 10. When it is called second time, r is created and initialized to 5 again. Now, the value of s becomes 20 since s initially held 10. Variable a is declared inside main() and can only be used inside main().

EXAMPLE

//Program to Demonstrate External variable

File: mul.cpp

```
int test=100; // assigning value to test
void multiply(int n)
{
  test=test*n;
}
```

File: main.cpp

```
#include<iostream>
#include "mul.cpp" // includes the content of sub.cpp
extern int test; // declaring test
int main()
{
  cout<<test<<endl;
  multiply(5);
  cout<<test<<endl;
  return 0;
}
```

Output

```
100
500
```

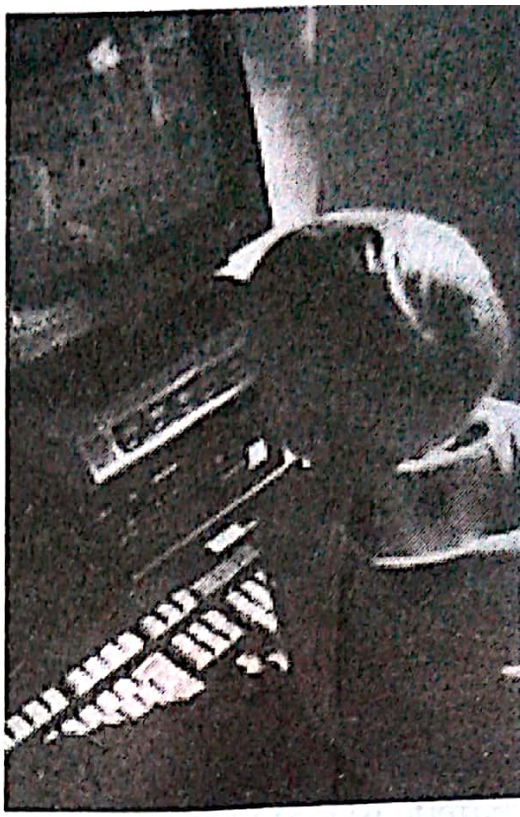
Self Assessment

Fill in the Blanks.

1. A function is a group of ..... that is executed when it is called from some point of the program.
2. The execution of the program starts from the function ....
3. The ..... specifies the type of the data the function returns.
4. The ..... list could be empty which means the function do not contain any parameters.
5. A function with default arguments should have values of ..... Arguments.
6. Macros are designed to reduce the.....overhead in case of small functions.
7. Inline functions are functions where the function body is expanded inline at the time of .....
8. Main drawback of macro is that it cannot handle arguments in the form of ..... properly.
9. Function overloading is a feature of C++ that allows us to create .....functions with the same name, so long as they have different parameters.
10. Overloaded functions have the same name but different number and.....
11. Main advantage of pass by reference is that it works much faster than return by value with larger constructs like ..... and classes.
12. One down side of return by reference is that we can't return ..... variables to the function by reference.

Exercise

1. What is function prototyping? Why is it necessary? When is it not necessary?
2. Discuss the concept of default arguments with suitable example. Also point out the situation in which we should be careful while assigning values to default arguments.
3. What is purpose of inline function?
4. Differentiate between the following:
   a. void main()
   b. int main()
   c. int main(int argn, char argv[])
5. Compare and contrast between inline functions and macros.
6. Write a program that uses overloaded member functions for converting temperature from Celsius to Kelvin scale.
7. Write a program to calculate the area of circle, rectangle and triangle using function overloading.
8. Do inline functions improve performance? Justify in your own word.
9. How can inline functions help with the tradeoff of safety vs. speed?
10. How can you use return by reference to return multiple values? Explain with proper example.



Chapter 5

Classes and Objects

Objectives

After studying this unit, you will be able to:

- Recognize how to specify a class.
- Define the member functions.
- Explain the creation of class objects.
- Access the class members.
- Discuss the access specifiers.

5.1 INTRODUCTION

Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into a single unit called classes. The data components of the class are called data members and the function components are called member functions. The data and functions of a class are intimately tied together. *Class is a blueprint of real world objects.* A programmer can create any number of objects of the same class. Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use. Classes are also referred to as programmer-defined data types.

It is important to note the subtle differences between a class and an object, here. *A class is a template that specifies different aspects of the object it models.* It has no physical existence. It occupies no memory. It only defines various members (data and/or methods) that constitute the class. *An object, on the other hand, is an instance of a class.* It has physical existence and hence occupies memory. You can think of a class as a data type; and it behaves just like as a data type like int, which does not have a physical existence and does not occupy any memory until a variable of that type is declared or created. In the same way, a class also does not exist physically and occupies no memory until an object of that class is created.

5.2 REVIEW OF STRUCTURES

Structure is an user defined data type which contains the collection of different types of data under the same name. A structure is compared with records in other languages.

Syntax:

```
struct  struct_name
{
        data-type var-name;
        data-type var-name;
        ..................
}; //end of structure
e.g.
struct currency
{
        int rs;
        float ps;
};
```

Declaring variable of structure

```
struct_name st_var_name;
e.g. currency c1,c2;
```

Initialization of structure variable:

structure variable can be initialized at the time of declaration as

```
currency c1 ={ 144,56.7};
```

However each member data of structure variable can be initialized separately after declaration as

```
currency c1;
c1.rs=144;
c1.ps=56.7;
```

But initialization all member data at once listing is illegal after declaration as

```
currency c1;
```

c1={144,56.7} // error:

EXAMPLE

```
//Using Structures
#include<iostream.h>
#include<conio.h> 0
struct currency
{
int rupees;
float paise;
}; // currency is name for struct currency
int main()
{
        currency c1,c3;
        currency c2 ={123,56.4};
        cout<<"Enter Rupees:"; cin>> c1.rupees;
        cout<<"Enter paises"; cin>> c1. paise;
        c3.paise = c1.paise+ c2.paise;
        if(c3.paise>=100.0)
        {
                c3.paise-=100.0 ;
                c3.rupees++;
        }
        c3.rupees+=c2.rupees+c1.rupees;
        cout<<"Rs." <<c1.rupees<<" Ps. " <<c1.paise<<" + ";
        cout<<"Rs." <<c2.rupees<<" Ps. "<<c2.paise<<" = ";
        cout<< "Rs."<<c3.rupees<<" Ps."<<c3.paise<<endl;
        getch();
        return 0;
}
```

Output

```
Enter Rupees:23
Enter paises34
Rs.23 Ps. 34 + Rs.123 Ps. 56.4 = Rs.185 Ps.90.4
```

C++ has made following three extensions to C structure, which makes the C++ structure more powerful than C structure.

- C++ Structures allows us to define functions as a member of structure

```
struct Employee
{
        ……
        ……
        void getdata( )
        {
```

```
        }
        ..................
        ..................
    }
```

- C++ structures allows us to use the access specifiers private and public inside the class

```
struct Employee
{
        public:
        int eid,sal;
        private:
        void getdata( )
        {
                //function body
        }
        ..................
}
```

- C++ structure allows us to use structures similar to that of primitive data types while defining variables.

```
struct Employee e;//C style
Employee e;      //C++ style
```

EXAMPLE

```
//C++ Structures
#include<iostream.h>
#include<conio.h>
struct Employee
{
    private:
    int eid,sal;
    public:
    void getdata( )
    {
        cout<<"Enter ID and Salary of an employee"<<endl;
        cin>>eid>>sal;
    }
    void display( )
    {
        cout<<"Emp ID:"<<eid<<endl<<"Salary:"<<sal<<endl;
    }
};
int main( )
{
    Employee e;
```

```
    e.getdata( );
    cout<<"Employee Details"<<endl;
    cout<<"----------------"<<endl;
    e.display( );
    getch( );
    return 0;
}
```

Output

```
Enter ID and Salary of an employee
Emp ID:101
Salary: 23000
Employee Details
101
23000
```

By default all members of structure has public visibility. Private members are only accessible inside structures but public members are accessible from anywhere in the program

5.3 CLASS AND OBJECT

A class is a template definition of the methods and variables in a particular kind of object. And an object is a specific instance of a class. It contains real values instead of variables. Think about classes, instances, and instantiation like baking a cake. A class is like a recipe for chocolate cake. The recipe itself is not a cake. You can't eat the recipe (or at least wouldn't want to). If you correctly do what the recipe tells you to do (instantiate it) then you have an edible cake. That edible cake is an instance of the chocolate cake class.

A class defines constituent members which enable these class instances to have state and behavior. Data field members (member variables or instance variables) enable a class object to maintain state. Other kinds of members, especially methods, enable a class object's behavior. In another word we can say that class is a user defined data type that contains member variables (attributes) and member functions (behavior) in the same unit. And objects are variables whose data type is some specified class. C++ structures can also include data members as well as member function inside it but classes are much more powerful than structures because it supports the concepts like inheritance, polymorphism, abstract classes etc.

5.3.1 Specifying a Class

The specification of class starts with the keyword *class* followed by the class name. Like structure, its body is delimited by braces terminated by a semicolon. The body of the class contains the keywords private, public, and protected (discussed later in inheritance). Private data and functions can only be accessed from within the member functions of that class. Public data or functions, on the other hand are accessible from everywhere. Usually the data within a class is private and functions are public. The data is hidden so it will be safe from accidental manipulation, while the functions that operated on the data are public so they can be accessed from outside the class.

The class also contains any number of data items and member functions. The general form of class declaration is:

```
class class_name
{
    private:
```

```
        data-type variable1;
        data-type variable2;
    data-type function1(argument declaration)
    {
        //Function body
    }
    data-type function2(argument declaration)
    {
        //Function body
    }
    .....
    .....
    public:
    data-type variable3;
    data-type variable4;
    data-type function3(argument declaration)
    {
        //Function body
    }

    data-type function4(argument declaration)
    {
        //Function body
    }
    .....
    };
```

For example, we can declare a class for rectangles as follows:

```
class rectangle
{
    private:
    int length;
    int breadth;
    public:
    void setdata(int l, int b)
    {
        length = l;
        breadth = b;
    }
    void showdata()
    {
        cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
    }
    int findarea()
```

```
    {
        return length * breadth;
    }
    int findperemeter()
    {
        return 2 * length * breadth;
    }
};
```

5.3.2 Creating Objects

The class declaration does not define any objects but only specifies what they will contain. Once a class has been declared, we can create variables (objects) of that type by using the class name (like any other built-in type variables). For example,

```
rectangle r;
```

It creates a variable (object) r of type rectangle. We can create any number of objects from the same class. For example,

```
rectangle r1, r2, r3;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brace. For example,

```
class rectangle
{
        .....
        .....
        .....
} r1, r2, r3;
```

Size of an object is equal to sum of size of all data members (instance variables) declared in the class.

5.3.3 Accessing Class Members

When an object of the class is created then the members are accessed using the '.' dot operator. For example,

```
r.setdata(4, 2);
r.showdata();
cout<<"Area = "<<r.findarea()<<endl;
cout<<"Peremeter = "<<r.findperemeter();
```

Private class members cannot be accessed in this way from outside of the class. For example, if the following statements are written inside the main function, the program generates compiler error.

```
r.length = length;
r1.breadth = breadth
```

EXAMPLE

```
//A Complete Program using class
#include<iostream.h>
#include<conio.h>
class rectangle
{
    private:
```

```
        int length; breadth;
        public:
        void setdata(int l, int b)
        {
                length = l;
                breadth = b;
        }
        void showdata()
        {
                cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
        }
        int findarea()
        {
                return length * breadth;
        }
        int findperemeter()
        {
                return 2 * length * breadth;
        }
};
int main()
{
        rectangle r;
        r.setdata(4, 2);
        r.showdata();
        cout<<"Area= "<<r.findarea()<<endl;
        cout<<"Peremeter= "<<r.findperemeter();
        getche();
        return 0;
}
```

Output

```
Length=4
Breadth=2
Area=8
Perimeter=12
```

5.4 DEFINING MEMBER FUNCTIONS OF THE CLASS

We can define member functions of the class in two ways: *inside the class and outside of the class.* In the first case both method declaration and definition is done at the same place inside the class. And in the second case method is declared inside the class but it's definition is provided outside of the class by using scope resolution operator. All the member functions defined inside the class are inline by default. But if the member functions defined inside the class are large and complex they are not expanded inline.

Defining member functions outside of the class is considered as good programming convention because it keeps class declaration separate from class definition. If we follow such programming convention, we can provide our class declaration to others by hiding actual logic of the program from them. The general form member functions defined outside of the class is:

```
return-type class-name :: function-name(argument declaration)
{
    //Function body
}
```

The symbol double colon(::) is called the binary scope resolution operator. For example, we can rewrite the same rectangle class as follows:

EXAMPLE

```
//Defining Member Functions
#include<iostream.h>
#include<conio.h>
class rectangle
{
    private:
    int length, breadth;
    public:
    void setdata(int l, int b)
    { //method definition inside the class
        length=l;
        breadth=b;
    }
    void showdata();
    int findarea();
    int findperimeter();
};
void rectangle :: showdata()
{ //method definition outside of the class
    cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
}
int rectangle :: findarea()
{
    return length * breadth;
}
int rectangle :: findperimeter()
{
    return 2 * (length + breadth);
}
int main()
{
    int a,p;
    rectangle r;
```

```
        r.setdata(4,5);
        r.showdata();
        a=r.findarea();
        cout<<"Area="<<a<<endl;
        p=r.findperimeter();
        cout<<"Perimeter="<<p<<endl;
        getch();
        return 0;
}
```

Output

```
Length=4
Breadth=5
Area=20
Perimeter=18
```

Functions defined outside the class are not normally inline. We can define a member function outside the class definition and still make it inline by just using the qualifier inline as follows:

```
inline void rectangle :: setdata(int l, int b)
{
        length = l;
        breadth = b;
}
```

5.5 MEMORY ALLOCATION FOR OBJECTS

For each object, the memory space for data members is allocated separately because the data members will hold different data values for different objects. However, all the objects in a given class use the same member functions. Hence, the member functions are created and placed in memory only once when they are defined as a part of a class specification and no separate space is allocated for member functions when objects are created.

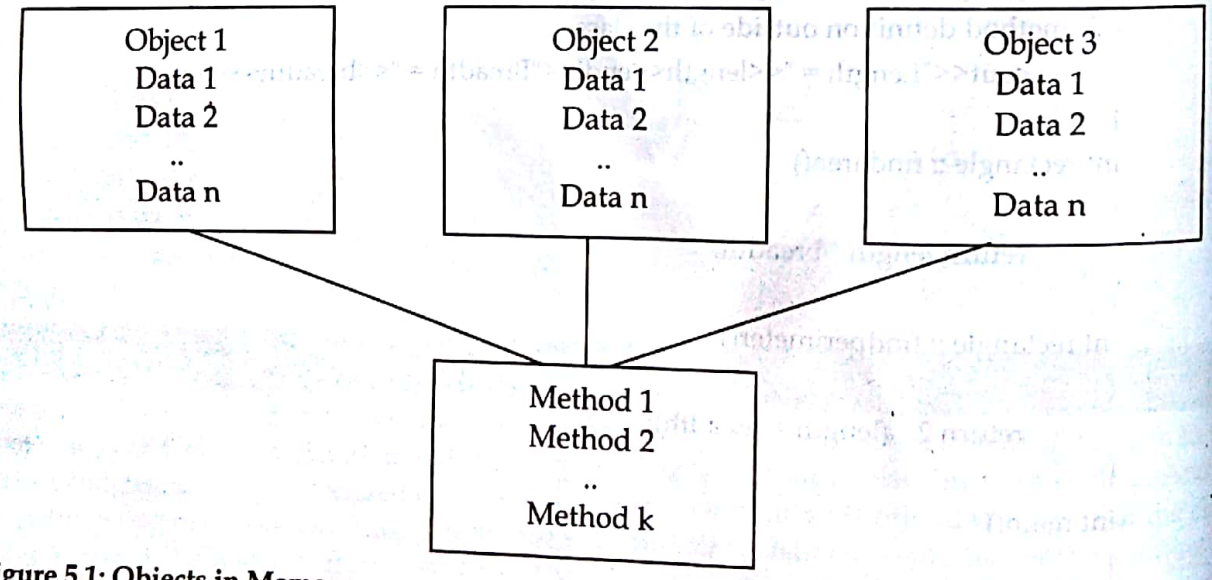


Figure 5.1: Objects in Memory

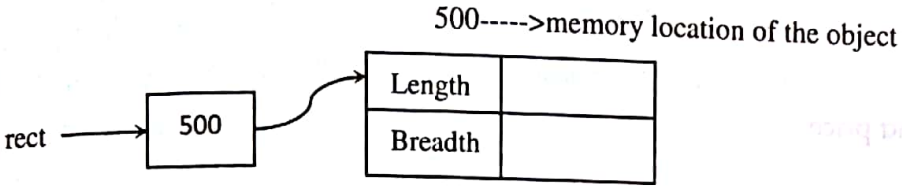
5.6 POINTER OBJECTS

It is perfectly valid to create pointers pointing to classes, in order to do that we must simply consider that once declared, the class becomes a valid type, so use the class name as the type for the pointer. For example:

```
rectangle *rect;
```

is a pointer to an object of class Rectangle. It only allocates memory to keep the address of an object of class rectangle. We should use new operator to allocate memory for the object of rectangle class as below:

```
*rect = new rectangle();
```



While creating pointer objects, you can also combine above two statements in single as below:

```
Rectangle *rect = new rectangle();
```

As it happens with structures, to refer directly to a member of an object pointed by a pointer you should use arrow operator (→). Here is an example

EXAMPLE

```
//Pointer to Classess
#include<iostream.h>
#include<conio.h>
class Item
{
    int code, price;
    public:
    void getdata()
    {
    cout<<"Enter code and price:"<<endl;
    cin>>code>>price;
    }

    void showdata()
    {
        cout<<"Code="<<code<<endl;
        cout<<"Price="<<price<<endl;
    }
};
int main()
{
    Item *a=new Item(); //pointer object
    Item b;
```

```
        a->getdata();
        b.getdata();
        cout<<"-----Item Details------"<<endl;
        cout<<"First Item:"<<endl;
        a->showdata();
        cout<<"Second Item:"<<endl;
        b.showdata();
        getch();
        return 0;
}
```

Output

```
Enter code and price
101
500
Enter code and price
102
700
----Item Details-----
First Item:
Code=101
Price=500
Second Item:
Code=102
Price=700
```

5.7 ARRAY OF OBJECTS

Any object, whether built-in or user-defined, can be stored in an array. When you declare the array, you tell the compiler the type of object to store and the number of objects for which to allocate room. The compiler knows how much room is needed for each object based on the class declaration. Accessing member data in an array of objects is a two-step process. You identify the member of the array by using the index operator ([ ]), and then you add the member access operator (.) or arrow (→) to access the particular member variable. Example given below demonstrates how you would create an array of five employees.

EXAMPLE

```
//Array of Objects
#include <iostream.h>
#include<conio.h>
const int MAX  =100;
class Employee
{
        private:
        int salary;
```

```
        int id;
        public:
        void getdata( )
        {
                cout <<"Enter the ID:";
                cin >> id;
                cout << "Enter the salary:";
                cin >> salary;
        }
        void putdata( )
        {
                cout <<"ID:" <<id<< "\tSalary:" << salary <<endl;
        }
};
int main()
{
        Employee e[MAX];
        int n=0;
        char ans;
        do
        {
                cout << "Enter the Employee Number::"<<n+1<<endl;
                e[n++].getdata();
                cout << "Enter another (y/n)?: " ;
                cin >> ans;
        } while ( ans != 'n' );
        cout<<endl<<"*******Employee Details*******"<<endl;
        for (int j=0; j<n; j++)
        {
                cout << "\nEmployee Number is:: " << j+1;
                e[j].putdata( );
        }
        getch();
        return 0;
}
```

Output

```
Employee number::1
Enter ID: 101
Enter Salary: 20000
Enter another(y/n)?y
Employee number::1
Enter ID: 102
```

```
Enter Salary: 30000
Enter another(y/n)?n
*******Employee Details*******
Employee number is::1  ID:101          Salary:20000
Employee number is::2  ID:102          Salary:30000
```

5.8 ACCESS SPECIFIERS

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. Keywords that are used to define access restriction to the class members are called access specifiers or access modifiers. C++ supports three success specifiers: *private, public, and protected*. A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

- **Private:** A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class would be private.
- **Public:** A public member variable or function is accessible from anywhere outside the class and within a program.
- **Protected:** A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes. We will learn derived classes and inheritance in later chapter.

EXAMPLE

```
//Access Specifiers and their visibility
#include<iostream.h>
#include<conio.h>
class Test
{
        private:
        int x;
        public:
        int y;
        void getdata()
        {
        cout<<"Enter x and y"<<endl;
        cin>>x>>y;
        }
        void display()
        {
        cout<<"x="<<x<<"y="<<y<<endl;
        }
};
int main()
{
      Test p;
```

```
    p.getdata();
    cout<<"Enter  value of x"<<endl;
    cin>>p.x; //generates error
    cout<<"Enter value of y"<<endl;
    cin>>p.y; //No error
    getch();
    return 0;
}
```

Here the statement "cin>>p.x;" generates error because x is private data member and is not accessible from outside of the class. But the statement "cin>>p.y;" does not generates error because y is public data member and is accessible from everywhere. Thus we can say that use of private access specifier is used to achieve data hiding in class. A member of a class can access any member of the same class independent of whether the member is public, protected or private.

5.9 NESTING OF MEMBER FUNCTIONS

A member function can be called by using its name from another member function of the same class. This is known as nesting of member functions. Like private data member, some situations may require certain member functions to be hidden from the outside call. A private member function can only be called by another function that is a member function of its class. In such situation nesting of member functions is important. For example, the class below shows the nesting of member function findinterest inside the member function findtotal.

EXAMPLE

```
//Nesting of Member functions
#include<iostream.h>
#include<conio.h>
class total
{
    private:
    float principle, time, rate;
    float findinterest()
    {
        return principle * time * rate / 100;
    }
    public:
    void setdata(float p, float t, float r)
    {
        principle = p;
        time = t;
        rate = r;
    }
    float findtotal()
    {
        return principle + findinterest();
    }
```

```
};
int main()
{
        float p,t,r,a;
        total ta;
        cout<<"Enter values of p, t, r"<<endl;
        cin>>p>>t>>r;
        ta.setdata(p,t,r);
        a=ta.findtotal();
        cout<<"Total Amount="<<a<<endl;
        getch();
        return 0;
}
```

Output

```
Enter values of p, t, r
2000
4
12
Total Amount=2960
```

5.10 OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as a function argument in three ways: *pass-by-value, pass-by-reference, and pass-by-pointer.*

5.10.1 Pass-by-value

If arguments are passed by value to the method, a copy of the object is passed to the function. Any changes made to the object inside the function do not affect the object used in the function call. Since all values of objects need to be copied into arguments of method invoked, it makes program slower when larger objects are used. An object can be passed by value to the function as below:

EXAMPLE

```
//Program to add two objects each having private data members feet and inches.
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
        int feet;
        int inches;
        public:
        void setdata(int f,float i)
        {
                feet=f;
                inches=i;
        }
```

```
        void adddistance(distance d1, distance d2)
        {
                feet = d1.feet + d2.feet;
                inches = d1.inches + d2.inches;
                feet=feet + inches/12;
                inches=inches%12;
        }
        void display()
        {
                cout<<"("<<feet<<", "<<inches<<")"<<endl;
        }
};
int main()
{
        distance d1, d2, d3;
        d1.setdata(5, 6);
        d2.setdata(7, 8);
        d3.adddistance(d1,d2);
        cout<<"d1 = ";
        d1.display();
        cout<<"d2 = ";
        d2.display();
        cout<<"d3 = ";
        d3.display();
        getch();
        return 0;
}
```

Output

```
d1 = (5, 6)
d2 = (7, 8)
d3 = (13, 2)
```

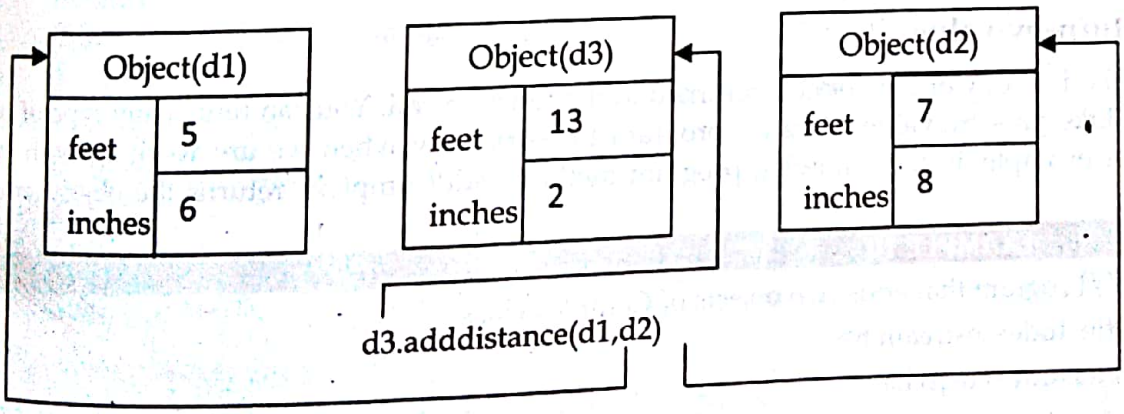


Figure 5.3: Representation of objects in memory.

5.10.2 Pass-by-reference

If arguments are passed by reference, an address of the object is passed to the function. The function works directly on the actual object used in the function call. This means that any changes made to the object inside the function will reflect in the actual object. Since only address of object is passed to the method invoked, it makes program faster when using larger objects. We can change above adddistance function for passing objects by reference as below:

```
void adddistance(distance &d1, distance &d2)
{
        feet = d1.feet + d2.feet;
        inches = d1.inches + d2.inches;
        feet=feet + inches/12;
        inches=inches%12;
}
```

This function must be called as follows:

```
d3.adddistance(d1,d2);
```

5.10.3 Pass-by-pointer

Like pass-by-reference method, pass-by-pointer method can also be used to work directly on the actual object used in the function call. For example,

```
void adddistance(distance* d1, distance *d2)
{
        feet = d1->feet + d2->feet;
        inches = d1->inches + d2->inches;
        feet=feet +inches/12
        inches= inches%12;
}
```

This function must be called as follows:

```
d3.adddistance(&d1,&d2);
```

5.11 RETURNING OBJECTS

A function not only receives objects as arguments but also can return them. A function can return objects also in three ways: *by value, by reference, and by pointer.*

5.11.1 Return-by-value

In this method, a copy of the object is returned to the function call. You can return any type of object by value. Like pass by value it makes program to work slow when we are working with larger objects. For example, in the following program method "addComplex" returns the object temp by value.

EXAMPLE 10

```
//Program that adds two objects of Complex class
#include<iostream.h>
#include<conio.h>
class Complex
{
```

```
        private:
        int real,img;
        public:
        void getdata( )
        {
                cout<<"Enter values of real and imaginary"<<endl;
                cin>>real>>img;
        }
        void display()
        {
                cout<<"("<<real<<"+i"<<img<<")"<<endl;
        }
        Complex addComplex(Complex c)
        {
                Complex temp;
                temp.real=real+c.real;
                temp.img=img+c.img;
                return temp;
        }
};
int main( )
{
        Complex c1,c2,c3;
        c1.getdata();
        c2.getdata();
        c3=c2.addComplex(c1);
        cout<<"Addition=";
        c3.display();
        getch();
        return 0;
}
```

Output

```
Enter values of real and imaginary
3
2
Enter values of real and imaginary
6
1
Addition=(9+i3)
```

5.11.2 Return-by-reference

In this method, an address of the object is returned to the function call. We cannot return local objects by reference because they goes out of scope when exiting from the method and hence no longer exists in memory. Like passing by reference, return by reference also makes program faster

while working with larger objects. For example, in the following program c is not an automatic obje and is returned by reference.

```
Complex& addComplex(Complex &c)
{
        c.real=real+c.real;
        c.img=img+c.img;
        return c;
}
```

This function must be called as follows:

```
c3 = c2.addcomplex(c1);
```

5.11.3 Return-by-pointer

Like return-by-reference method, return-by-pointer returns address of the object to the function cal It also cannot return local objects and makes program to work faster in case of larger objects. Fo example, in the following function c is returned by pointer.

```
Complex* addComplex(Complex *c)
{
        c->real=real+c->real;
        c->img=img+c->img;
        return c;
}
```

In the main function, c3 must be declared as a pointer variable and the function call must be made a below:

```
Complex c1, c2, *c3;
c3=c2.addComplex(&c1);
cout<<"Addition = ";
c3->display();
```

5.12 STATIC DATA MEMBERS

If a data member in a class is defined as static, then only one copy of that member is created for th entire class and is shared by all the objects of that class, no matter how many objects are created Hence, these data members are normally used to maintain values common to the entire class and ar also called class variables. Memory for static data members is allocated at the time of declaration. Al static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class using the scope resolution operator (::) to identify which class it belongs to.

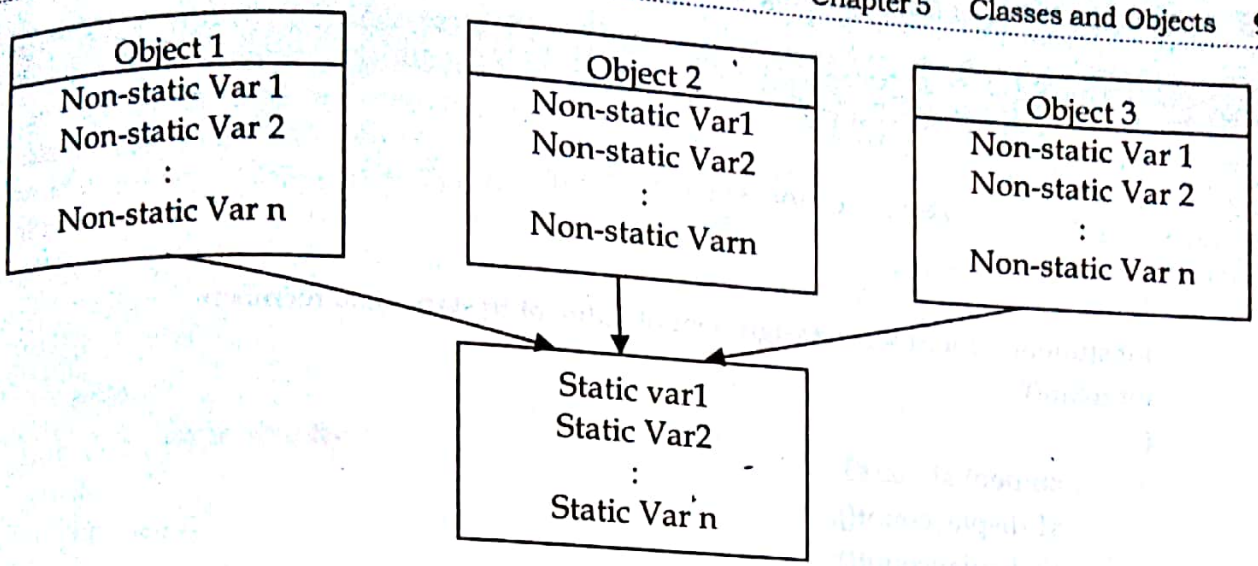


Figure 5.4: Memory Allocations for Static data members.

The linker will report an error if a static data member is declared but not defined. The definition must occur outside the class, and only one definition is allowed. Thus, it is common to put it in the implementation file for the class. The syntax sometimes gives people trouble, but it is actually quite logical. For example, if you create a static data member inside a class like this:

```
class A
{
        static int i;
public:
 //...
};
```

Then you must define storage for that static data member in the definition file like this:

```
int A::i = 1;
```

EXAMPLE

```
//Program for counting number of objects
#include<iostream.h>
#include<conio.h>

class student
{
        private:
        int roll;
        char name[20];
        static int count; //static data member
        public:
        void getdata()
        {
                cout<<"Enter roll and name"<<endl;
                cin>>roll>>name;
                count++;
```

```
    }
    void displaycount()
    {
            cout<<"count:"<<count<<endl;
    }
};
int student :: count=0;//Assignment of value of to static data members
int main()
{
    student s1, s2, s3;
    s1.displaycount();
    s2.displaycount();
    s3.displaycount();
    s1.getdata();
    s2.getdata();
    s3.getdata();
    s1.displaycount();
    s2.displaycount();
    s3.displaycount();
    getche();
    return 0;
}
```

Output

```
count:0
count:0
count:0
Enter roll and name
1  Ram
Enter roll and name
2  Hari
Enter roll and name
3  Rina
count:3
count:3
count:3
```

In this program, the static variable count is initialized to zero when the objects are created. The count is incremented whenever data is supplied to an object. Since the data is supplied three times, the variable count is incremented three times. Because there is only one copy of the count shared by all the three objects, all the three calls to the displaycount. member function cause the value 3 to be displayed.

Since memory for static data member is allocated at the point of declaration and that memory is shared by all objects of the class, they are also called class level variables or non-instance variables.

5.13 STATIC MEMBER FUNCTIONS

You can also create static member functions that work for the class as a whole rather than for a particular object of a class. When you create a static member function, you are expressing an association with a particular class. You can call a static member function in the ordinary way, with the dot or the arrow, in association with an object. However, it's more typical to call a static member function by using class itself along with scope resolution operator, like this:

```
class X
{
public:
        static void f(){};
};
int main()
{
        X::f(); //Calling static member function
}
```

A static member function cannot access ordinary data members, only static data members. It can call only other static member functions. Normally, the address of the current object (this pointer) is passed in implicitly when any member function is called, but a static member has no this, which is the reason it cannot access ordinary members. Thus, you get the tiny increase in speed afforded by a global function because a static member function doesn't have the extra overhead of passing this. At the same time you get the benefits of having the function inside the class.

EXAMPLE

```
//Modification of above program static method for displaying count
#include<iostream.h>
#include<conio.h>
class student
{
        private:
        int roll;
        char name[20];
        static int count; //static data member
        public:
        void getdata()
        {
                cout<<"Enter roll and name"<<endl;
                cin>>roll>>name;
                count++;
        }
        static void displaycount()
        {
                cout<<"count:"<<count<<endl;
        }
};
int student :: count=0;
```

```
int main()
{
        student s1, s2, s3;
        student::displaycount();
        s1.getdata();
        s2.getdata();
        s3.getdata();
        student::displaycount();
        getch();
        return 0;
}
```

Output

```
count:0
Enter roll and name
1  Ram
Enter roll and name
2  Hari
Enter roll and name
3  Rina
count:3
```

5.14 NESTED CLASSES

A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables. Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class also have no special access to members of a nested class. The following example demonstrates this:

```
class A
{
        int x;
        public:
        class B
        {
                int y;
        void f(A* p, int i)
        {
                p->x = i;
/* The compiler cannot allow this statement because A::x  is private:*/
        }
}; //end of class B
void g(B *p)
```

```
    {
        int z = p->y;
        /*The compiler cannot allow this statement because C::y is private*/
        }
};
```

The compiler would not allow the statement p->x = i because A::x is private. The compiler also would not allow the statement int z = p->y because C::y is private.

You can access the static members x and y and member functions of the nested class by using a qualified type name. Qualified type names allow you to define a typedef to represent a qualified class name. You can then use the typedef with the :: (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
class outside
{
    public:
  class nested
  {
     public:
    static int x;
    int fun( );
    };
};
int outdside::nested::x = 5;
typedef outside::nested inc
int inc::fun( )
{
 return 0;
};
```

It is better to define the class as nested if it is only used by outer enclosing class. Doing this also makes encapsulation strong because nested can be hidden from client programmers.

EXAMPLE

```
//Complete Program showing use of  nested class
#include<iostream.h>
#include<conio.h>
class Employee
{
    public:
    void getdata()
    {
        cout<<"Enter Id, Name and Salary of an employee"<<endl;
        cin>>eid>>ename>>salary;
        cout<<"Enter Date of Birth of an Employee"<<endl;
        db.getDOB();
    }
```

```
        void display()
        {
                cout<<"EID:"<<eid<<endl;
                cout<<"Name:"<<ename<<endl;
                cout<<"Salary:"<<salary<<endl;
                cout<<"DOB:";
                db.displayDOB();
        }
        class DOB
        {
                int y,m,d;
                public:
                void getDOB()
                {
                        cout<<"Enter Year:";
                        cin>>y;
                        cout<<"Enter Month:";
                        cin>>m;
                        cout<<"Enter Day:";
                        cin>>d;
                }
                void displayDOB()
                {
                        cout<<y<<"-"<<m<<"-"<<d<<endl;
                }
        };
        private:
        int eid, salary;
        char ename[20];
        DOB db;
};
int main()
{
        Employee e;
        e.getdata();
        cout<<"######Employee Details#######"<<endl;
        e.display();
        getch();
        getch();
}
```

Output

Enter Id, Name and Salary of an employee

```
101 Binek 30000
Enter Date of Birth of an Employee
Enter Year:1985
Enter Month:5
Enter Day:13
######Employee Details#######
EID:101
Name:Binek
Salary:30000
DOB:1985-5-13
```

5.15 LOCAL CLASSES

A class that is declared within a function definition is called local class. In practice we use local class rarely. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

```
int x;                  // global variable
void fun()
{
   static int y;
/* static variable y can be used by local class*/

   int x;
/*auto variable x cannot be used by local class */

   extern int z
   class local     // local class
   {
      int a( ) { return x; }
      // error, local variable x cannot be used by a

      int b( ) { return y; }
      // valid, static variable y

      int c( ) { return ::x; }
               // valid, global x

      int d( ) { return z; }
               // valid, extern variable z
   };
}
int main()
{
   local* z;     // error: the class local is not visible
```

.....

}

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword inline. We cannot define static data members in local class because there is no way of defining static data members at global scope.

5.16 CONST OBJECT AND CONST MEMBER FUNCTIONS

Some objects need to be modified and some do not. We can use the keyword const to specify that an object is not modifiable and that any attempt to modify the object should result compiler error. The statement

```
const distance d1(5, 6.7);
```

declares a constant object d1 of class distance and initializes it to 5 feet and 6.7 inches. These objects must be initialized. A const object can only invoke a const member function. If a member function does not alter any data in the class, then we may declare and define it as a const member function as follows:

```
void display()const
{
    cout<<"("<<feet<<", "<<inches<<")"<<endl;
}
```

The qualifier const is inserted after the function's parameter list in both declarations and definitions. The compiler will generate an error message if such functions try to alter the data values. A const member function cannot call a non-const member function on the same class.

Inside a class const allocates storage within each object and represents a value that is initialized once and then cannot change. The use of const inside a class means "This is constant for the lifetime of the object. Thus, when you create an ordinary (non-static) const inside a class, you cannot give it an initial value. This initialization must occur in the constructor, of course, but in a special place in the constructor because a const must be initialized at the point it is created.

EXAMPLE

```
//Constant member variables
#include<iostream.h>
#include<conio.h>
class Fred
{
const int size;
public:
    Fred(int sz): size(sz)
    {}
    void print()
    {
        cout << size << endl;
    }
};
int main()
```

```
{
        Fred a(1), b(2), c(3);
        a.print(), b.print(), c.print();
        getch();
        return 0;
}
```

Output

```
1
2
3
```

However, each different object may contain a different value for that constant. The above use of const is interesting and probably useful in cases, but it does not create a compile-time constant inside a class. The answer requires the use of an additional keyword, static. The static keyword means there is only one instance, regardless of how many objects of the class are created.

5.17 FRIEND FUNCTIONS

There were three levels of internal protection for the different members of a class: public, protected and private. The concepts of data hiding and encapsulation dictate that private and protected members of a class cannot be accessed from outside the class. That is non-member functions of a class cannot access the non-public members (data members and member functions) of a class. However, we can achieve this by using friend functions. The friend functions allow operations between two different classes. Generally the use of friend functions is out of an object-oriented programming methodology because it violates the concept of data hiding. So whenever possible it is better to use members of the same class. To make an outside function friendly to a class, we simply declare the function as a friend of the class as shown below:

EXAMPLE

```
//Friend function
#include<iostream.h>
#include<conio.h>
class sample
{
        int a;
        int b;
        public:
void setvalue()
{
        a = 25;
        b = 40;
}
friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2;
```

```
}
int main()
{
        sample x;
        x.setvalue();
        cout<<"Mean value = "<<mean(x);
        getch();
        return 0;
}
```

Output

Mean value = 32.5

Some special characteristics of friend functions are:

1. Since, it is not in the scope of the class to which it has been declared as a friend, it cannot be called using the object of the class.
2. It can be invoked like a normal function without the help of any object.
3. Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
4. It can be declared either in the public or the private part of a class without affecting its meaning.
5. Usually, it has the objects as arguments.

Furthermore, a fried function also acts as a bridging between two classes. For example, if we want a function to take objects of two classes as arguments and operate on their private members, we can inherit the two classes from the same base class and put the function in the base class. But if the classes are unrelated, there is nothing like a friend function. For example,

EXAMPLE

```
//More on Frient Functions
#include<iostream.h>
#include<conio.h>
class beta;
class alpha
{
        private:
        int data;
        public:
        void setdata(int d)
{
                data = d;
        }
        friend int sum(alpha, beta);
};
class beta
{
        private:
```

```
        int data;
        public:
        void setdata(int d)
{
                data = d;
        }
        friend int sum(alpha, beta);
};
int sum(alpha a, beta b)
{
        return a.data + b.data;
}
int main()
{
        alpha a;
        a.setdata(7);
        beta b;
        b.setdata(3);
        cout<<"sum="<<sum(a, b);
        getch();
        return 0;
}
```

Output

Sum=10

5.18 FRIEND CLASSES

A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

EXAMPLE

```
//Friend Class
#include<iostream.h>
#include<conio.h>
class alpha
{
        private:
        int x;
        public:
        void setdata(int d)
        {
                x = d;
        }
        friend class beta;
```

```
};
class beta
{
public:
void func(alpha a)
{
        cout<<a.x<<endl;
}
};
int main()
{
        alpha a;
        a.setdata(99);
        beta b;
        b.func(a);
        getch();
        return 0;
}
```

Output

99

5.19 THIS POINTER

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object on which this function was called. The pointer 'this' acts as an implicit argument to all the member function apart from the explicit arguments passed to it. When a number of objects are created from the same class each object's data members are created as separate copies. However, only a single copy of methods is retained in the memory. That is all objects of a class share a single copy of the compiled class functions. A particular object is referenced by "this" pointer internally. This pointer may be accessed from within a member function as this. Since it is a pointer, dereferencing it will return the entire invoking object. We can use this pointer for following two purposes:

- To resolve the name conflict between member variables and local variables in a method
- To return the invoking object

EXAMPLE

```
//This Pointer
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person
{
        char name[20];
        int age;
        public:
        void setdata(char name[],int age)
```

```
    {
        strcpy(this->name,name); //name conflict resoltion
        this->age=age;
    }
    void display()
    {
        cout<<"Name::"<<this->name<<endl;
        cout<<"Age::"<<this->age<<endl;
    }
    person isElder(person p)
    {
        if(age>p.age)
        return *this; //returning invoking object
        else
        return p;
    }
};
int main()
{
    person p,p1,p2;
    p1.setdata("Aayan",1);
    p2 .setdata("Binek",2);
    p=p1.isElder(p2);
    cout<<"Elder one is:"<<endl;
    p.display();
    getch();
    return 0;
}.
```

Output

```
Elder one is:
Name::Binek
Age::2
```

Self Assessment

Fill in the Blanks.

1. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into a single unit called .....
2. Class also does not exist physically and occupies no memory until an ..... of that class is created.
3. Private data and functions can only be accessed from within the ..... of that class.
4. Defining member functions outside of the class is considered as good programming convention because it keeps class declaration separate from .....
5. To refer directly to a member of an object pointed by a pointer you should use ..... operator

6. If arguments are passed by value to the method, ... of the object is passed to the function.
7. In case of pass by reference, since only address of object is passed to the method invoked, it makes program faster when using ..... objects,
8. Static data members are normally used to maintain values common to the entire class and are also called ..... variables.
9. We can call a static member function in the ordinary way, with the dot or the arrow, in association with an object. However, it's more typical to call a static member function by using ....., without any specific object, using the scope-resolution operator,
10. It is better to define the class as nested if it is only used by outer enclosing class. Doing this also makes..... strong because nested can be hidden from client programmers.
11. We can not define static data members in local class because there is no way of defining static data members at ..... scope.
12. A const member function cannot call a ..... member function on the same class.
13. The concepts of data hiding and encapsulation dictate that private and protected members of a class cannot be accessed from outside the class. However, we can achieve this by using .....
14. C++ uses a unique keyword called ..... to represent an object that invokes a member function.

Exercise Questions

1. How data hiding can be achieved in class? Discuss with example.
2. Write a program to add two objects of class Complex having data members real and imaginary.
3. What are the different ways of defining member functions?which one of them is better way and why?
4. Differentiate between pointer variable of class type and pointer object and show the use of pointer object with example of your choice.
5. How constant defined by #define directive are different from constants defined by const keyword. Analyze,
6. Write a program to read and display 10 objects of Item class containing data members item name, code and price.
7. Does friend function violate data hiding? Explain with an example. What are the some advantages or disadvantages of using friend functions?
8. Why static data members are important? Explain it concept along with static member functions, with suitable example.
9. Describe different methods of returning objects from function with their merits and demerits.
10. In which situation it is better to use nested classes? Explain with suitable example. Also describe the concept of local classes
11. Explain the uses of this pointer with suitable example.
12. Write a program to create a class account with data members account_no, account holders name, balance and minimum balance. Add member functions create_account, deposit, withdraw and balance_inquary to the class. Create an object of the account class and show all operation with this.

❑❑❑

Chapter 6

Constructors and Destructors

Objectives

After studying this unit, you will be able to:

- Recognize the need for constructor and destructors.
- Describe the copy constructor.
- Explain the dynamic constructor.
- Discuss the destructors.
- Explain the constructor and destructors with static members.

6.1 INTRODUCTION

When an object is created all the members of the object are allocated memory spaces. Each object has its individual copy of member variables. However the data members are not initialized automatically. If left uninitialized these members contain garbage values. Therefore it is important that the data members are initialized to meaningful values at the time of object creation. When a C++ program runs, it invariably creates certain objects in the memory and when the program exits the objects must be destroyed so that the memory could be reclaimed for further use. C++ provides mechanisms to cater to the above two necessary activities through constructors and destructors methods.

6.2 CONSTRUCTORS

A constructor is a special member function that is executed automatically whenever an object is created. It is used for automatic initialization. Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function. The name of the constructor is same as the class name. For example,

```
class rectangle
{
        private:
        int length;
        int breadth;
        public:
        rectangle()
{       //constructor
                length = 0;
                breadth = 0;
        }
                .....
                .....
};
```

Constructors have some special characteristics. These are:

1. Constructors should be defined or declared in the public section.
2. They do not have return types.
3. They cannot be inherited but a derived class can call the base class constructor.
4. Like functions, they can have default arguments.
5. Constructors cannot be virtual
6. We cannot refer to their addresses
7. An object with a constructor (or destructor) cannot be used as a member of a union.
8. They make 'implicit calls' to the new and delete operators when a memory allocation is required.

6.3 TYPES OF CONSTRUCTORS

6.3.1 Default Constructor

The constructor used in above example is default constructor. A constructor that accepts no parameters is called default constructor. If a class does not include any constructor, the compiler

supplies a default constructor. The default constructor method is called automatically at the time of creation of an object and does nothing more than initializing the data variables of the object to valid initial values. If we create an object by using the declaration

```
rectangle r1;
Default constructor is invoked
```

Actually the default constructor method has been defined in *system.object* class. Since every class that you create is an extension of *system.object* class, this method is inherited by all the classes.

6.3.2 Parameterized Constructors

Parameterized constructors are used if it is necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called 'Parameterized constructors.' For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type int.

```
class rectangle
{
        private:
        int length;
        int breadth;
        public:
        rectangle(int l, int b)
{    //parameterized constructor
                length = l;
                breadth = b;
        }
                .....
};
```

We can use parameterized constructors in two ways: by calling the constructor explicitly and by calling the constructor implicitly (sometimes called shorthand method). The declaration

```
rectangle r1 = rectangle(5, 6.7);
```

illustrates the first method of calling and the declaration

```
rectangle r1(5, 6.7);
```

illustrates the second method of calling.

Note that C++ selects one constructor by matching the signature of the method being called. Also, once you define a constructor method, the default constructor is overridden and is not available to the class. Therefore you must also define a constructor method resembling the default constructor method having no parameters.

6.3.3 Copy Constructor

A copy constructor method allows an object to be initialized with another object of the same class. It implies that the values stored in data members of an existing object can be copied into the data variables of the object being constructed, provided the objects belong to the same class. A copy constructor has a single parameter of reference type that refers to the class itself as shown below:

```
rectangle(rectangle& r)
{
```

```
        lentht = r.length;
        breadth = r.breadth;
    }
```

Once copy constructor is defined, we can use copy constructor as below:

```
rectangle r2(r1);
```

It creates new object r2 and performs member-by-member copy of r1 into r2. Remember that we cannot pass the argument by value to a copy constructor.

6.3.4 Default Copy Constructor

If you don't define copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. But it will do shallow copy, if class contains pointers than default copy constructor cannot copy value pointed by pointer variable of one object into pointer variable of another object rather both of them will point to the same place. To handle this situation we need to create copy constructor. Default copy constructor can be invoked as below:

```
rectangle r2 = r1;
```

The process of initializing through assignment operator is also known as copy initialization. C++ has overloaded assignment operator which invokes copy constructor so that it copies the member values of one object to member values of another object.

6.4 CONSTRUCTOR OVERLOADING

We can define more than one constructor in a class either with different number of arguments or with different type of argument which is called constructor overloading. It is a variation of function overloading and is used to achieve polymorphism.

EXAMPLE

```
//Constructor overloading
#include<iostream.h>
#include<conio.h>
class Item
{
    int code, price;
    public:
    Item()
    {//Default Constructor
        code= price =0;
    }
    Item(int c,int p)
    {//Parameterized Constructor
        code=c; price=p;
    }
    Item(Item &x)
    {//Copy Constructor
        code=x.code;
        price= x.price;
    }
```

```
        void display()
        {
        cout<<"Code::"<<code<<endl<<"Price::"<<price<<endl<<endl;
        }
};
int main()
{
        Item I1;
        Item I2(102,300);
        Item I3(I2);
        I1.display();
        I2.display();
        I3.display();
        getch();
        return 0;
}
```

Output

```
Code:0
Price:0

Code:102
Price:300

Code:102
Price:300
```

6.5 DYNAMIC CONSTRUCTORS

Allocation of Memory during the creation of objects can be done by the constructors too. The memory is saved as it allocates the right amount of memory for each object (Objects are not of the same size). Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. New operator is used to allocate memory. The following program concatenates two strings. The constructor function initializes the strings using constructor function which allocates memory during its creation.

EXAMPLE

```
//Dynamic Constructor
#include <iostream.h>
#include<conio.h>
#include <string.h>
class String
{
char * name;
int length;
public:
String ()
```

```
{
        length = 0;
        name = new char [length = 1];
}
String (char *s)
{
        length = strlen (s);
        name = new char [length + 1];
        strcpy(name,s );
}
void display (void)
{
        cout<<"\n Name :-"<<name;
}
void join (String & a, String & b)
{
        length = a.length + b.length;
        delete name;
        name = new char [length + 1];
        strcpy (name,a.name);
        strcat (name,"");
        strcat (name,b.name);
}
};
int main()
{
        String fn("Aayan");
        String ln("Saud");
        String n;
        n.join(fn,ln);
        n.display();
        getch();
        return 0;
}
```

Output

Name:- Aayan Saud

6.6 CONSTRUCTORS WITH DEFAULT ARGUMENTS

This method is used to initialize object with user defined parameters at the time of creation. Consider the following Program that calculates simple interest. It declares a class interest representing principal, rate and year. The constructor function initializes the objects with principal and number of years. If rate of interest is not passed as an argument to it the Simple Interest is calculated taking the default value of rate of interest.

EXAMPLE

```
//Constructor with default arguments
#include <iostream.h>
#include <conio.h>
class interest
{
        int principal, rate, year;
        float amount;
        public:
        interest (int p, int n, int r=10);
        void cal (void);
};
interest::interest (int p, int n, int r)
{
        principal = p; year = n; rate = r;
}
void interest::cal (void)
{
        cout<<"Principal="<<principal<<endl;
        cout <<"Rate="<<rate<<endl;
        cout<<"Year="<<year<<endl;
        amount = principal+(float) (principal*year*rate)/100;
        cout<<"Amount="<<amount<<endl;
}
int main ( )
{
        interest i1(1000,2);
        interest i2(1000, 2,15);
        i1.cal();
        i2.cal();
        getch();
        return 0;
}
```

Output

```
        Principal=1000
        Rate=10
        Year=2
        Amount=1200
Principal=1000
Rate=15
Year=2
Amount=1300
```

6.7 DESTRUCTORS

A destructor is a special member function that is executed automatically just before lifetime of an object is finished. The most common use of destructors is to destroy the memory that was allocated for the object by the constructor. When a program no longer needs an instantiated object it destroys it. If you do not supply a destructor function, C++ supplies a default destructor for you, unknown to you. The program uses the destructor to destroy the object for you. In many cases you do not need a destructor function. However, if your class created dynamic objects, then you need to define your own destructor in which you will delete the dynamic objects. This is because dynamic objects cannot be deleted by default destructor. Therefore, to delete the dynamic object you should override the default destructor. A destructive method has the following characteristics:

- A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde (~).
- Like constructors, destructors do not have a return value.
- Destructors take no arguments. Hence, we can use only one destructor in a class.
- Destructors can be virtual
- Destructors cannot be inherited

EXAMPLE

```
//Program showing destructor calling
#include<iostream.h>
#include<conio.h>
class Test
{
        private:
        int x,y;
        public:
        Test()
        {
                x=0;
                y=0;
                cout<<"Memory is allocated"<<endl;
        }
        ~Test()
        {
                cout<<"x="<<x<<"\t"<<"y="<<y<<"\t"<<endl;
                cout<<"Memory is deallocated"<<endl;
        }
};
int main()
{

        {
                Test p;
        }//life time of p finishes here, and destructor is called
```

```
    getch();
    return 0;
}
```

Output

```
Memory is allocated
x:0    y:0
Memory is de-allocated
```

Whenever you create an object using new keyword, you must explicitly destroy it using delete keyword, failing which the object would remain homed in the memory and in the course of program execution there may come a time when sufficient memory is not available for creation of the more objects. This phenomenon is referred to as memory leak. Programmers must consider memory leak seriously while writing programs for the obvious reasons. Objects in an array are destructed in the reverse order of their creation. This means the object created at first will be destroyed at last.

Self Assessment

Fill in the Blanks.

1. When a C++ program runs it invariably creates certain objects in the memory and when the program exits the objects must be destroyed so that the memory could be ..... for further use.
2. A constructor is a special member function that is executed automatically whenever an ..... created.
3. If a class does not include any constructor, the compiler supplies a..... constructor.
4. Once you define a constructor method, the default constructor is overridden and is not available to the class. Therefore you must also define a ..... resembling the default constructor method having no parameters.
5. C++ has overloaded ..... operator which invokes copy constructor so that it copies the member values of one object to member values of another object
6. Allocation of memory to objects at the time of their construction is known as ..... construction of objects.
7. If your class created dynamic objects, then you need to define your ....., in which you will delete the dynamic objects.
8. Objects in an array are destructed in the ..... order of their creation. This means the object created at first will be destroyed at last.
9. A copy constructor has a single parameter of ..... type that refers to the class itself.
10. Since every class that you create is an extension of ..... class, default constructor method is inherited by all the classes.

Exercise Questions

1. Write a program to calculate prime number using constructor.
2. Is there any difference between List x; and List x();? Explain.
3. Can one constructor of a class call another constructor of the same class to initialize the this object? Justify your answers with an example.
4. Why cannot we pass an object by value to a copy constructor?

5. Write a program to show constructor overloading. Also explain how a compiler identifies the function to be called.
6. What is the purpose of having default arguments in a constructor? Explain with suitable examples.
7. How do you destroy objects created by using new operator? Explain with example. Also explain what happens if you do not destroy such object?
8. Spot out the error in the following code and correct it.

```
class one
{
        int x,y;
        public:
        one(int);
        one(int);
        one(int, int);
        ~one(int)
        {
        cout << "Object being destroyed!!";
        }
}
```

9. How is a copy constructor different from a constructor? Illustrate with suitable examples.
10. Debug the following code:

```
class interest
{
        int principal, rate, year;
        float am;
        public:
        interest(int p=1000, int n, int r = 10);
};
interest::interest(int p=1000, int n, int r = 10)
{
        principal = p; year = n; rate = r;
}
```

❑❑❑

Chapter 7

Operator Overloading

After studying this unit, you will be able to:

- Recognize the operator overloading.
- Describe the rules for operator overloading.
- Explain the overloading of unary operators.
- Discuss the various binary operators with friend function and member function
- Understand data conversion.

7.1 INTRODUCTION

C++ provides a rich collection of operators. You have already seen the meaning and uses of many such operators in previous units. C++ also incorporates the option to use language standard operators between classes in addition to between fundamental types by using the concept of operator overloading. For example:

```
int a, b, c;
a = b + c;
```

is perfectly valid, since the different variables of the addition are all fundamental types. But if we try to use operators with objects (user defined types) compiler will produce an error. That in principle is not valid between non-fundamental types. Operator overloading is one of the most exciting features of C++. The term operator overloading refers to giving the normal C++ operators additional meaning and semantics so that we can use them with user-defined data types. For example, C++ allows us to add two variables of user-defined types with the same syntax that is applied to the basic type. We can overload all the C++ operators except the following:

- **Class member access/dot operators (.)**
- **Scope resolution operator (::)**
- **Size operator (sizeof)**
- **Conditional operator (?:)**

Generally the operators that can't be overloaded because overloading them could cause serious program errors or it is syntactically not possible, For instance the sizeof operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you cannot overload it with your own runtime code. It is syntactically not possible to do. Scope resolution and member access operators work on names rather than values. C++ has no syntax for writing code that works on names rather than values so syntactically these operators cannot be overridden. And overloading the conditional operator does not have any useful purpose.

Although the semantics of an operator can be extended, we cannot change its syntax and semantics that govern its use such as the number of operands, precedence, and associativity. Operator overloading is done with the help of a special function, called operator function. The general form of an operator function is:

```
return-type operator op(arguments)
{
Function body
}
```

Where return-type is the type returned by the operation, operator is the keyword, and op is the operator symbol. For example, to add two objects of type distance each having data members feet of type int and inches of type float, we can overload + operator as follows:

```
distance operator +(distance d2)
{
    //Function body
}
```

And we can call this operator function with the same syntax that is applied to its basic types as follows:

```
d3 = d1 + d2;
```

Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary

operators, while a member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. We cannot use friend functions to overload certain operators. These operators are:

- Assignment operator (=)
- Function call operator (())
- Subscripting operator ([ ])
- Class member access/arrow operator (→)

When overloading an operator using a member function:

1. The leftmost operand of the overloaded operator must be an object of the class type.
2. The leftmost operand becomes the implicit parameter (this pointer). All other operands become function parameter.

By using operator overloading we can easily access the objects to perform any operations.

Rules for Overloading Operators

To overload any operator, we need to understand the rules applicable Let us revise some of them which have already been explored.

1. Operators already predefined in the C++ compiler can be only overloaded. Operator cannot change operator templates that is for example the increment operator ++ is used only as unary operator. It cannot be used as binary operator.
2. Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But, still when we it with two numbers, it finds sum of numbers.
3. Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).
4. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
5. Overloaded operators must either be a non-static class member function or a global function.
6. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type.

7.2 OVERLOADING UNARY OPERATORS

An operator which acts upon only one operator is called unary operators. In case of unary operator overloaded using a member function no argument is passed to the function whereas in case of operator overloading by using a friend function a single argument must be passed.

7.2.1 Overloading Prefix operator

Let us consider the increment (++) operator. This operator increases the value of an operand by 1 when applied to a basic data item. Similarly, the decrement (--) operator decreases the value of an operand by 1 when applied to a basic data item. We can overload these operator so that it can be used to increase or decrease the value of each data member when applied to an object.

EXAMPLE

```
//Overloading pre-increment operator
#include<iostream.h>
#include<conio.h>
```

```
class rectangle
{
        private:
        int length;
        int breadth;
        public:
        rectangle(int l, int b)
        {
                length = l;
                breadth = b;
        }
        void operator ++()
        {
                ++length;
                ++breadth;
        }
        void display()
        {
                cout<<"Length = "<<length<<endl
                  <<"Breadth = "<<breadth;
        }
};

int main()
{
        rectangle r1(5, 6);
        ++r1;    //equivalent to r1.operator ++();
        r1.display();
        getch();
        return 0;
}
```

Output

```
Length = 6
Breadth = 7
```

7.2.2 Overloading Postfix Operator

To overload the postfix operator we need to use *int* as dummy argument to the operator function. That int isn't really an argument, and it doesn't mean integer. It is simply a signal to the compiler to create the postfix version of the increment and decrement operator. We can overload postfix operator as follows:

```
void operator ++(int)
{
        length++;
        breadth++;
```

```
}
```

We can call this operator function as follows:

```
r1++; //equivalent to r1.operator++();
```

7.2.3 Overloading Unary Operator using Friend Function

It is also possible to overload unary operators using a friend function. Since no object is used to invoke the friend function, no object is passed to the friend function implicitly. Therefore when overloading unary operator by using friend function, it must take an object as argument. You can overload pre-increment operator using friend function as follows:

EXAMPLE

```
//Overloading increment Operator using friend function
#include<iostream.h>
#include<conio.h>
class rectangle
{
        private:
        int length, breadth;
        public:
        rectangle(int l, int b)
        {
                length = l;         breadth = b;
        }
        friend void operator ++(rectangle&);
        void display()
        {
        cout<<"Length = "<<length<<endl <<"Breadth = "<<breadth;
        }
};
void operator ++(rectangle& r)
{
        ++r.length;
        ++r.breadth;
}
int main()
{
        rectangle r1(5, 6);
        ++r1;    //operator ++(r1);
        r1.display();
        getch();
        return 0;
}
```

Output

Length = 6
Breadth = 7

7.2.4 Overloading Negation Operator

Minus is the operator which can be binary as well as unary. If we write expression like "z=x-y", minus (-) operator acts as binary operator. But if we write expression like "x=-y", minus (-) operator acts as unary operator which makes value of y negative and puts that value in y. You can overload negation (-) operator to make value of object negative as follows:

EXAMPLE

```
//Overloading Negation Opertaor
#include<iostream.h>
#include<conio.h>
class point
{
        int x, y;
      public:
        void getdata()
        {
                cout<<"Enter x and y coordinate"<<endl;
                cin>>x>>y;
        }
        void display()
        {
                cout<<"("<<x<<","<<y<<")"<<endl;
        }
        point operator-()
        {
                point t;
                t.x=-x;          t.y=-y;
                return t;
        }};
int main()
{
        point p,q;
        p.getdata();
        q=-p;
        cout<<"q=";
        q.display();
        getch();
        return 0;
}
```

Output

```
Enter x and y coordinate
3 7
q= (-3,-7)
```

7.3 OVERLOADING BINARY OPERATORS

Binary Operators are operators, which require two operands to perform the operation. When they are overloaded by means of member function, the function takes one argument whereas it takes two arguments in case of operator overloading by using friend function.

7.3.1 Overloading Plus Operator

Let us consider the addition (+) operator. This operator adds the values of two operands when applied to a basic data item. This operator can be overloaded to add the values of corresponding data members when applied to two objects. For example,

EXAMPLE

```
//Overloading plus operator
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
        int feet;
        int inches;
        public:
        void getdata( )
        {
                cout<<"Enter feet and inch"<<endl;
                cin>>feet>>inches;
        }
        distance operator +( distance d2)
        {
                distance d3;
                d3.feet = feet + d2.feet;
                d3.inches = inches + d2.inches;
                d3.feet=d3.feet +d3.inches/12;
                d3.inches=d3.inches%12;
                return d3;
        }
        void display()
        {
                cout<<"("<<feet<<", "<<inches<<")"<<endl;
        }
};
```

```
int main()
{
        distance d1,d2,d3;
        d1.getdata();
        d2.getdata();
        d3 = d1 + d2;    //d1.operator +(d2);
        cout<<"d1 = ";
        d1.display();
        cout<<"d2 = ";
        d2.display();
        cout<<"d3 = ";
        d3.display();
        getche();
        return 0;
}
```

Output

```
d1 = (5, 6)
d2 = (7, 8)
d3 = (13, 2)
```

7.3.2 Overloading plus Operator using Friend Function

It is also possible to overload binary operators using a friend function. Here we need to use two arguments in operator function because friend functions are called without using objects and hence no object is passed as implicitly to the function. You can overload plus operator using friend function as follows:

EXAMPLE

```
//Overloading binary operator using friend function
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
        int feet, inches;
        public:
        distance()
        {
                feet=inches=0;
        }
        distance(int f, int in )
        {
                feet=f;                    inches=in;
        }
        friend distance operator +(distance, distance);
        void display()
```

```
    {
        cout<<"("<<feet<<", "<<inches<<")"<<endl;
    }
};
distance operator +( distance x,distance y)
{
    distance r;
    r.feet = x.feet + y.feet;
    r.inches = x.inches + y.inches;
    r.feet=r.feet +r.inches/12;
    r.inches=r.inches%12;
    return r;
}
int main()
{
    distance d1(5, 6), d2(7, 8), d3;
    d3 = d1 + d2;    //d1.operator +(d2);
    cout<<"d1 = ";
    d1.display();
    cout<<"d2 = ";
    d2.display();
    cout<<"d3 = ";
    d3.display();
    getche();
    return 0;
}
```

Output

```
d1 = (5, 6)
d2 = (7, 8)
d3 = (13, 2)
```

7.3.3 Overloading Comparison Operator

Overloading comparison operator is almost similar to overloading plus (+) operator except that it must return value of an integer type. This is because result of comparison is always true or false. C++ treats true and non-zero value and false as zero. You can overload < (less than) operator to compare two objects as follows:

EXAMPLE

```
//Overloading comparision (less than) operator
#include<iostream.h>
#include<conio.h>
class Time
{
    int hr;
    int min;
```

```
        public:
        void getdata()
        {
                cout<<"Enter hour and minute"<<endl;
                cin>>hr>>min;
        }
        int operator <(Time t)
        {
                int ft, st;//first time and second time
                ft=hr*60+min;//convert into minute
                st=t.hr*60+t.min;
                if(ft<st)
                        return 1;
                else
                        return 0;
        }
};
int main()
{
        Time t1,t2;
        t1.getdata();
        t2.getdata();
        if(t1<t2)
        cout<<"t1 is less than t2"<<endl;
        else
        cout<<"t1 is greater or equal to t2"<<endl;
        getch();
        return 0;
}
```

Output

1st run

Enter hour and minute

4 45

Enter hour and minute

5 25

t1 is less than t2

2nd run

Enter hour and minute

2 45

Enter hour and minute

3 30

t1 is greater or equal to t2

3rd run

```
Enter hour and minute
2 45
Enter hour and minute
2 50
t1 is less than t2
```

7.3.4 Overloading + Operator to Concatenate two Strings

You can overload + operator to concatenate two strings as follows:

EXAMPLE

```
//String Concatenation
#include<iostream.h>
#include<string.h>
#include<conio.h>
class String
{
        char *s;
        int l;//length of string
        public:
        void getdata()
        {
                char str[20];
                cout<<"Enter a string"<<endl;
                cin>>str;
                l=strlen(str);      s=new char[l+1];
                strcpy(s,str);
        }
        void display()
        {
                cout<<s<<endl;
        }
        String operator+(String x)
        {
                String temp;
                temp.s=new char[l+x.l+1];
                strcpy( temp.s,s);          strcat(temp.s, x.s);
                return temp;
        }
};
int main()
{
        String s1,s2,s3;
        s1.getdata();
        s2.getdata();
```

```
        s3=s1+s2;
        cout<<"s3=";
        s3.display();
        getch();
        return 0;
}
```

Output

```
Enter String
1970
Enter string
AD
S3=1970AD
```

7.4 NAMELESS TEMPORARY OBJECTS

We can also use nameless temporary objects to add two objects whose purpose is to provide a return value for the function. For example,

```
distance operator + ( distance d2)
{
        int ft = feet + d2.feet;
        int  in = inches + d2.inches;
        ft=ft+in/12;
        in=int%12;
        return (distance(ft, in));    //an unnamed temporary object
}
```

We use nameless temporary objects when the object is not needed to be referenced in future again.

7.5 OVERLOADING ASSIGNMENT OPERATOR

We can overload assignment (=) operator in c++. By overloading assignment operator, all values of one object can be copied to another object by using assignment operator. Assignment operator must be overloaded by a non-static member function only. If the overloading function for the assignment operator is not written in the class, the compiler generates the function to overload the assignment operator.

```
//Example: Program to overload assignment operator
#include<iostream>
class Marks
{
        private:
        int m1;
        int m2;
        public:
        //Default constructor
        Marks()
        {
                m1 = 0;
                m2 = 0;
```

```
        }
        // Parametrised constructor
        Marks(int i, int j)
        {
                m1 = i;
                m2 = j;
        }
        // Overloading of Assignment Operator
        void operator=(const Marks &M )
    {
        m1 = M.m1;
        m2 = M.m;
    }
    void Display()
    {
        cout << "Marks in 1st Subject:" << m1;
        cout << "Marks in 2nd Subject:" << m2;
    }
};
int main()
{
    // Make two objects of class Marks
    Marks mark1(45, 89);
    Marks mark2(36, 59);
    cout << " Marks of first student : ";
    Mark1.Display();
    cout << " Marks of Second student :";
    Mark2.Display();
    // use assignment operator
    Mark1 = Mark2;
    cout << " Mark in 1st Subject :";
    Mark1.Display();
    return 0;
}
```

Output

```
Marks of first student :
Mark in 1st Subject : 45
Marks in 2nd Subject : 89

Marks of Second student :
Mark in 1st Subject : 36
Marks in 2nd Subject : 59

Marks of First student :
```

Mark in 1st Subject : 36
Marks in 2nd Subject : 59

7.6 TYPE CONVERSION

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type conversion. A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler. Explicit type conversions (type casting) are used when a programmer want to get around the compiler's typing system. If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routines by ourselves. Four types of situations might arise in the data conversion in this case.

1. Conversion from basic type to another basic type
2. Conversion from basic type to user defined type
3. Conversion from user defined type to basic type
4. Conversion from one user defined type to another user defined type

Conversion from basic type to another basic type is already discussed in chapter 2. This can be automatic (implicit) type conversion or explicit type conversion (type casting).

7.6.1 Conversion from basic type to user defined type

To convert basic types to user defined type (object) it is necessary to use the constructor. The constructor in this case takes single argument whose type is to be converted. For example,

EXAMPLE

```
//Basic to object conversion
#include<iostream.h>
#include<conio.h>
class distance
{
        private:
        int feet;
        int inch;

        public:
        distance(int f,int i)
        {
                feet=f;
                inch=i;
        }
        distance(float m)
        {
                feet = int(m);
                inch = 12 * (m - feet);
        }
        void display()
        {
                cout<<"Feet = "<<feet<<endl <<"Inch = "<<inch;
```

```
    }
};
int main()
{
    float f = 2.5;
    distance d = f;
    d.display();
    getch();
    return 0;
}
```

Output

```
Feet = 2
Inches = 6
```

7.6.2 Conversion from user defined type to basic type

To convert user defined types (objects) to basic type it is necessary to overload cast operator. Remember that overloaded cast operator does not have return type. Its implicit return type is the type to which object need to be converted. To convert object to basic type, we use conversion function as below:

EXAMPLE

```
//Object to basic conversion
#include<iostream.h>
#include<conio.h>
class distance
{
    private:
    int feet;
    int inch;
    public:
    distance(int f,int i)
    {
        feet=f;
        inch=i;
    }

    operator float()
    {
        float a= feet + inch/12.0;
        return a;
    }
};
int main()
{
```

```
    distance d(8, 6);
    float x = (float)d;
    cout<<"x = "<<x;
    getch();
    return 0;
}
```

Output

x = 8.5

Dynamic cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

7.6.3 Conversion from user defined type to another user defined type

This type of conversion can be carried out either by a constructor or an operator function. It depends upon where we want the routine to be located – in the source class or in the destination class.

Routine in the Source Class

If we want to convert objects of one class to object of another class, it is necessary that the operator function be placed in the source class. For example,

EXAMPLE

```
//Object to Object Conversion (Method in Source Class)
#include<iostream.h>
#include<conio.h>
class distance
{
    int feet;
    int inch;
    public:
    distance()
    {
        feet=inch=0;
    }

    distance(int f, int i)
    {
        feet = f;
        inch = i;
    }
    void display()
    {
        cout<<feet<<"ft "<<inch<<"inch"<<endl;
    }
};

class dist
{
```

```
    int meter;
    int centimeter;
    public:
    dist(int m, int c)
    {
            meter = m;
            centimeter = c;
    }
    operator distance( ) //operator function
    {
            distance d;
            int f,i;
            f= meter*3.3;
            i=centimeter*0.4;
            f=f+i/12;
            i=i%12;
            return distance(f,i);
     }
};
int main()
{
      distance d1;
      dist d2(4,50);
      d1=d2;
      d1.display();
      getche();
      return 0;
}
```

Output

14ft 8inch

Routine in the Destination Class

In this case, it is necessary that the constructor be placed in the destination class. This constructor is a single argument constructor and serves as an instruction for converting the argument's type to the class type of which it is a member. For example,

EXAMPLE

```
//Object to Object Conversion (Method in destination Class)
#include<iostream.h>
#include<conio.h>
class distance
{
        int meter;float cm;
        public:
        distance(int m, int c)
```

```
        {
                meter = m;        cm = c;
        }
        int getmeter()
        {
                return meter;
        }
        float getcentimeter()
        {
                return cm;
        }
};
class dist
{
        int feet;
        int inch;
        public:
        dist()
        {
                feet=inch=0;
        }
        dist(int f, int i)
        {
                feet = f;
                inch = i;
        }
        dist(distance d)
        {
                int m,c;
                m=d.getmeter();
                c=d.getcentimeter();
                feet= m*3.3;
                inch= c*0.4;
                feet=feet+inch/12;
                inch= inch%12;
        }
        void display()
        {
                cout<<feet<<"ft  "<<inch<<"inch"<<endl;
        }
};
int main( )
{
```

```
    distance d1(6,40);
    dist d2=d1;
    d2.display();
    getche();
    return 0;
}
```

Output

20ft 4inch

Self Assessment

Fill in the Blanks.

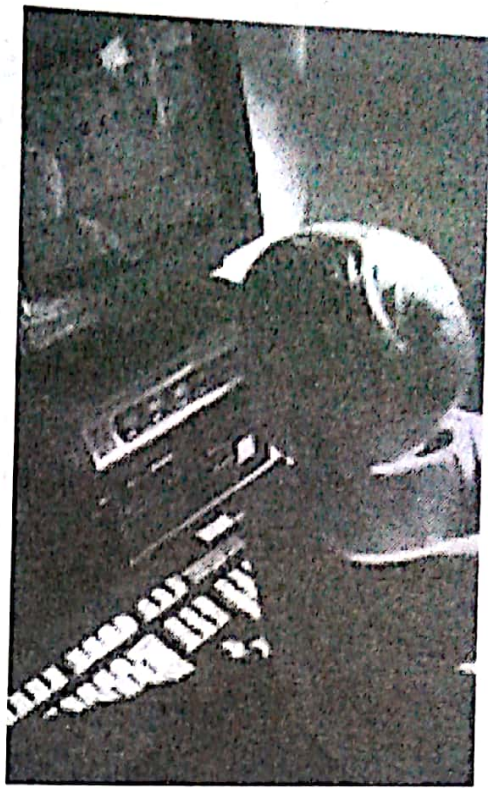
1. The term operator overloading refers to giving the normal C++ operators additional ..... so that we can use them with user-defined data types.
2. Operator overloading is done with the help of a special function, called ..... function.
3. When overloading an operator using a member function, the leftmost operand becomes the ..... parameter (this pointer).
4. Unary operators, overloaded by means of a member function, take ..... explicit argument and return ..... explicit values.
5. To overload the post-increment operator we need to use ..... as dummy argument to the operator function.
6. In case of binary operator overloading we need to use ..... arguments in operator function because friend functions are called without using objects and hence no object is passed as implicitly to the function.
7. Overloading comparison operator is almost similar to overloading plus (+) operator except that it must return value of an ..... type. This is because result of comparison is always true or false.
8. We use ..... temporary objects when the object is not needed to be referenced in future again.
9. To convert class types (objects) to basic type it is necessary to overload ..... operator.
10. If we want to convert objects of one class to object of another class, it is necessary that the operator function be placed in the ..... class.

Exercise Questions

1. Overload the addition operator (+) to assign binary addition.
2. Which operators are not allowed to be overloaded?
3. What are the differences between overloading a unary operator and that of a binary operator? Illustrate with suitable examples.
4. Why is it necessary to convert one data type to another? Illustrate with suitable examples.
5. How many arguments are required in the definition of an overloaded unary operator?
6. When used in prefix form, what does the overloaded + + operator do differently from what it does in postfix form?

7. Overload ++ operator to change a string from lower case to upper case and overload - operator to change string from upper case to lower case
8. Write a note on unary operators.
9. What are the various rules for overloading operators?
10. The assignment operations cause automatic type conversion between the operand as per certain rules. Describe.
11. There are three types of situations that arise where data conversion are between incompatible types. What are three situations explain briefly.
12. Write a program to convert an object of Dollar class to object of Ruppes class. Assume that Dollar class has data member's dol and cent and Rupees class have data member's rs and paisa.

❑❑❑



Chapter 8

Inheritance

Objectives

After studying this unit, you will be able to:

- Recognize the inheritance.
- Describe the different types of inheritance.
- Explain the ambiguity in multiple and multipath inheritance.
- Discuss the virtual base class.
- Identify the overriding member function.
- Demonstrate the execution of constructor and destructor with inheritance.

8.1 INTRODUCTION

Inheritance (or derivation) is the process of creating new classes, called **derived classes**, from existing classes, called **base classes**. The derived class inherits all the properties from the base class and can add its own properties as well. The inherited properties may be hidden (if private in the base class) or visible (if public or protected in the base class) in the derived class.

Inheritance uses the concept of code **reusability**. Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance. Reusing existing code saves time and money and increases program's reliability. An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

8.1.1 Defining Derived Class

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class derivedclassname : access_specifier baseclassname
{
.....
..... // members of derived class
};
```

The colon (:) indicates that the *derivedclassname* class is derived from the *baseclassname* class. The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the accessibility status of derived features. For example,

```
class xyz //base class
{
//members of xyz
};
class ABC: public xyz //public derivation
{
//members of ABC
};
class ABC : XYZ //private derivation (by default)
{
//members of ABC
};
```

Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

8.2 FORMS OF INHERITANCE

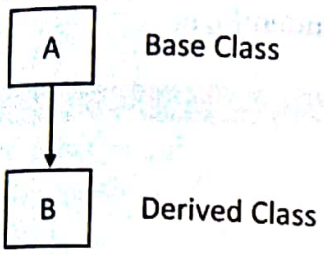
A class can inherit properties from one or more classes and from one or more levels. On the basis of this concept, there are five forms of inheritance.

1. Single inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance

4. Multilevel Inheritance
5. Hybrid Inheritance

8.2.1 Single Inheritance

In single inheritance, a class is derived from only one base class. The example and figure below show this inheritance.



Implementation Skeleton

```
class A
{
        //members of A
};
class B : public A
{
        //members of B
};
```

EXAMPLE

```
//Sigle Inheritance
#include <iostream.h>
#include<conio.h>
class Student
{
        char *name;
        int age;
        public:
        Student(char *n, int a)
        {
                name = n;
                age = a;
        }
        char *getName()
        {
                return name;
        }
        void display()
        {
                cout<<"Name::"<<name<<endl;
                cout<<"Age::"<<age<<endl;
        }
```

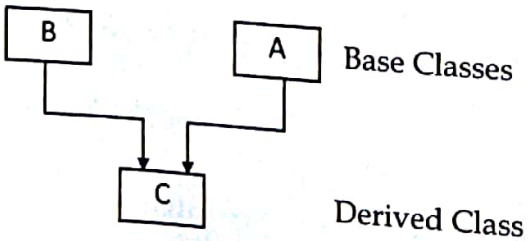
```
};
class ForeignStudent : public Student
{
        char *country;
        public:
        ForeignStudent(char *n, int a, char *c) : Student(n, a)
        {
                country = c;
        }
        void displayForeign()
        {
                display();
                cout<<"Country::"<<country<<endl;
        }
};
int main()
{
        ForeignStudent fs("Steven", 21, "UK");
        fs.displayForeign();
        getch();
        return 0;
}
```

Output

```
Name::Steven
Age::21
Country::UK
```

8.2.2 Multiple Inheritance

In this inheritance, a class is derived from more than one base class. The example and figure below show this inheritance.



Implementation Skeleton

```
class A
{
        //members of A
};
class B
{
```

```
        //members of B
};
class C : public A, public B
{
        //members of C
};
```

EXAMPLE

```
//Multiple Inheritance
#include<iostream.h>
#include<conio.h>
class Teacher
{
        int tid;
        char subject[20];
        public:
        void getTeacher()
        {
                cout<<"Enter Teacher id and subject"<<endl;
                        cin>>tid>>subject;
        }
        void displayTeacher()
        {
                cout<<"Teacher ID::"<<tid<<endl;
                        cout<<"Subject::"<<subject<<endl;
        }
};
class Staff
{
        int sid;
        char level[10];
        public:
        void getStaff()
        {
                cout<<"Enter staff ID and level"<<endl;
                cin>>sid>>level;
        }
        void displayStaff()
        {
                cout<<"Staff ID:"<<sid<<endl;
        cout<<"Level::"<<level<<endl;
        }
};
```

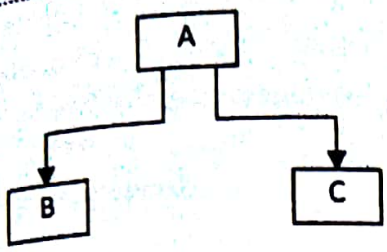
```
class coordinator: public Teacher, public Staff
{
char program[10];
public:
void getdata()
        {
                getTeacher();
                getStaff();
                cout<<"Enter Program"<<endl;
                cin>>program;
        }
        void displaydata()
        {
                displayTeacher();
                displayStaff();
                cout<<"Program::"<<program;
        }
};
int main()
{
        coordinator c;
        c.getdata();
        cout<<"-----Coordinator details------"<<endl;
        cout<<"--------------------------"<<endl;
        c.displaydata();
        getch();
        return 0;
}
```

Output

```
Enter Teacher ID and Subject
101  MIS
Enter Staff ID and level
503  Fifth
-----Coordinator details------
--------------------------
Teacher ID:: 101
Subject::MIS
Staff ID::503
Level::Fifth
```

8.2.3 Hierarchical Inheritance

In this type, two or more classes inherit the properties of one base class. The example and figure below show this inheritance.



Base Class

Derived Classes

Implementation Skeleton

```
class A
{
        //members of A
};
class B
{
        //members of B
};
class C : public A, public B
{
        //members of C
};
```

EXAMPLE

```
//Hierarchical Inheritance
#include<iostream.h>
#include<conio.h>
class Employee
{
        int eid, salary;
        public:
        void getEmp()
        {
                cout<<"Enter id and salary of employee"<<endl;
                cin>>eid>>salary;
        }
        void displayEmp()
        {
                cout<<"Emp ID::"<<eid<<endl;
                cout<<"Salary::"<<salary<<endl;
        }
};
class Engineer:public Employee
{
        char dept[10];
        public:
        void getdata()
```

```
        {
                getEmp();
                cout<<"Enter Department"<<endl;
                cin>>dept;
        }
        void display()
        {
                displayEmp();
                cout<<"Departement::"<<dept<<endl;
        }
};
class Typist:public Employee
{
        int ts;//typing speed
        public:
        void getdata()
        {
                getEmp();
                cout<<"Enter typing speed"<<endl;
                cin>>ts;
        }
        void display()
        {
                displayEmp();
                cout<<"Typing Speed::"<<ts<<endl;
        }
};
int main()
{
        Engineer e;        Typist t;
        e.getdata();       t.getdata();
        cout<<"------Employee Details-------"<<endl;
        cout<<"-----------------------"<<endl;
        e.display();       cout<<endl;
        t.display();
        getch();
        return 0;
}
```

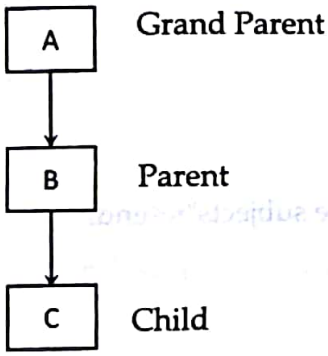
Output

```
Enter id and salary of employee
201   30000
Enter Department
```

```
Construction
Enter id and salary of employee
209   15000
Enter typing speed
67
-----Employee Details--------
---------------------------

Emp ID::201
Salary::30000
Departement::construction

Emp ID::209
Salary::15000
Typing Speed::67
```

2.4 Multilevel Inheritance

The mechanism of deriving a class from another derived class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels. The example and figure below show this inheritance.



mplementation Skeleton

```
class A
{
      //members of A
};
class B : public A
{
      //members of B
};
class C : public B
{
      //members of C
};
```

EXAMPLE

```
//Multilevel Inheritance
#include<iostream.h>
#include<conio.h>
```

```
class Student
{
        int roll;
        char name[20];
        public:
        void getStudent()
        {
                cout<<"Enter roll number and name of student"<<endl;
                cin>>roll>>name;
        }
        void displayStudent()
        {
                cout<<"Roll Number::"<<roll<<endl;
                cout<<"Name::"<<name<<endl;
        }
};
class Marks: public Student
{
        int sub1, sub2, sub3;
        public:
        void getMarks()
        {
                cout<<"Enter Marks in three subjects"<<endl;
                cin>>sub1>>sub2>>sub3;
        }
        void displayMarks()
        {
                cout<<"Sujecct1:"<<sub1<<endl;
                cout<<"Subject2:"<<sub2<<endl;
                cout<<"Subject3:"<<sub3<<endl;
        }
        int findTotalMarks()
        {
                return sub1+sub2+sub3;
        }
};
class Result: public Marks
{
        float total,percentage;
        public:
        void getdata()
        {
                getStudent();
```

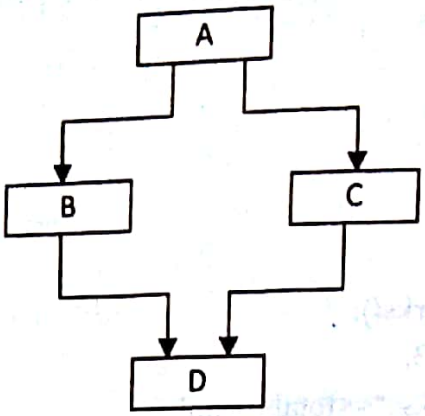
```
            getMarks();
        }
        void displaydata()
        {
                displayStudent();
                displayMarks();
                total=findTotalMarks();
                percentage=total/3;
                cout<<"Total Marks::"<<total<<endl;
                cout<<"Percentage::"<<percentage;
        }
};
int main()
{
        Result r;
        r.getdata();
        cout<<"-----Result details------"<<endl;
        cout<<"----------------------"<<endl;
        r.displaydata();
        getch();
        return 0;
}
```

Output

```
Enter roll number and name of student
3
Ayan
Enter Marks in three subjects
89
95
87
-----Result details------
----------------------
Roll Number::3
Name::Ayan
Sujecct1:89
Subject2:95
Subject3:87
Total Marks::271
Percentage::90.3333
```

8.2.5 Hybrid Inheritance

This type of inheritance includes more than one type of inheritance mentioned previously. The example and figure below show this inheritance.



Implementation Skeleton

```
class A
{
        //members of A
};

class B : public A
{
        //members of B
};
class C : public A
{
        //members of C
};
class D : public B, public C
{
        //members of D
};
```

8.3 PROTECTED ACCESS SPECIFIER

If a class member is private, it can be accessed only within the class where it lies. However, if a class member is public, it can be accessed from within the class and from outside the class as well. Therefore if we want to access members from derived class we must make members public. This violates the concept of data hiding which is the heart of object oriented programming. A protected member, on the other hand, can be accessed from within the class where it lies and from any class derived from this class. It can't be accessed from outside of these classes. Thus a protected member achieves data hiding as well as allows data member to be accessed from derived class. The table below summarizes this concept.

Access Specifier	Accessible from Own Class	Accessible from Derived Class	Accessible from Objects Outside the Class
Public	Yes	yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

EXAMPLE

```
//Program to prove visibility
#include<iostream.h>
#include<conio.h>
class Base
{
        private:
        int x;
        protected:
        int y;
        public:
        int z;
};
class Derived: public Base
{
        public:
        void getdata()
        {
                cout<<"Enter x, y and z"<<endl;
                cin>>x;
                //invalid, generates error because x is private
                cin>>y; cin>>z;
        }
};
int main()
{
        Derived d;
        d.getdata();
        cout<<"x="<<d.x<<endl;
        //invalid, generates error because x is private

        cout<<"y="<<d.y<<endl;
        //invalid, generates error because x is protected

        cout<<"z="<<d.z<<endl;//valid
        getch();
        return 0;
}
```

In the above program the statement "cin>>x" inside getdata() function of Derived class generates error because data member 'x' is private and is not accessible from subclass. But the statement "cin>>y" does not generate error because y is protected and is accessible from subclass also.

Again the statements "cout<<"x="<<d.x<<endl" and "cout<<"y="<<d.y<< endl inside the main function generates error because x and y are private and protected respectively and hence are not

accessible from main function. But the statement "cin>>z" inside getdata() function of derived class and the statement "cout<<"z="<<d.y<<endl" inside main function does not generate error because y is public and is accessible from everywhere.

EXAMPLE

```
//Multilevel inheritance by using protected access specifier
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
        int roll;
        char *name;
        public:
        Student()
        {
                name=NULL;
                roll=-1;
        }
        Student(int r, char *n)
        {
                name=new char[strlen(n)+1];
                roll=r;
                name=n;
        }
        void displayStudent()
        {
                cout<<"Name::"<<name<<endl;
                cout<<"Roll::"<<roll<<endl;
        }
};
class Marks:public Student
{
        protected:
        int sub1,sub2,sub3;
        public:
        Marks()
        {
                sub1=sub2=sub3=0;
        }
        Marks(int r, char *n,int x,int y,int z):Student(r,n)
        {
                sub1=x;
                sub2=y;
```

```
        sub3=z;
    }
    void displayMarks()
    {
        cout<<"Subject1::"<<sub1<<endl;
        cout<<"Subject2::"<<sub2<<endl;
        cout<<"Subject3::"<<sub3<<endl;
    }
};
class Result:public Marks
{
    int total;
    float percentage;
    public:
    Result()
    {
        total=0;
        percentage=0;
    }
    Result(int r,char *n,int x,int y,int z):Marks(r,n,x,y,z)
    {
        total=sub1+sub2+sub3;
        percentage=float(total)/3;
    }
    void displayResult()
    {
        displayStudent();
        displayMarks();
        cout<<"Total marks::"<<total<<endl;
        cout<<"Percdentage::"<<percentage<<endl;
    }
};
int main()
{
    Result r(1,"Roshan",56,87,92);
    cout<<"-------Result Details--------"<<endl;
    cout<<"------------------------------"<<endl;
    r.displayResult();
    getch();
    return 0;
}
```

Output

```
------Result Details--------
-------------------------
Name::Roshan
Roll::1
Subject1::56
Subject2::87
Subject3::92
Total marks::235
Percdentage:: 78.333336
```

8.4 PUBLIC, PROTECTED AND PRIVATE INHERITANCE

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form is:

```
class derived-class-name : visibility-mode base-class-name
{
        Members of derived classes
};
```

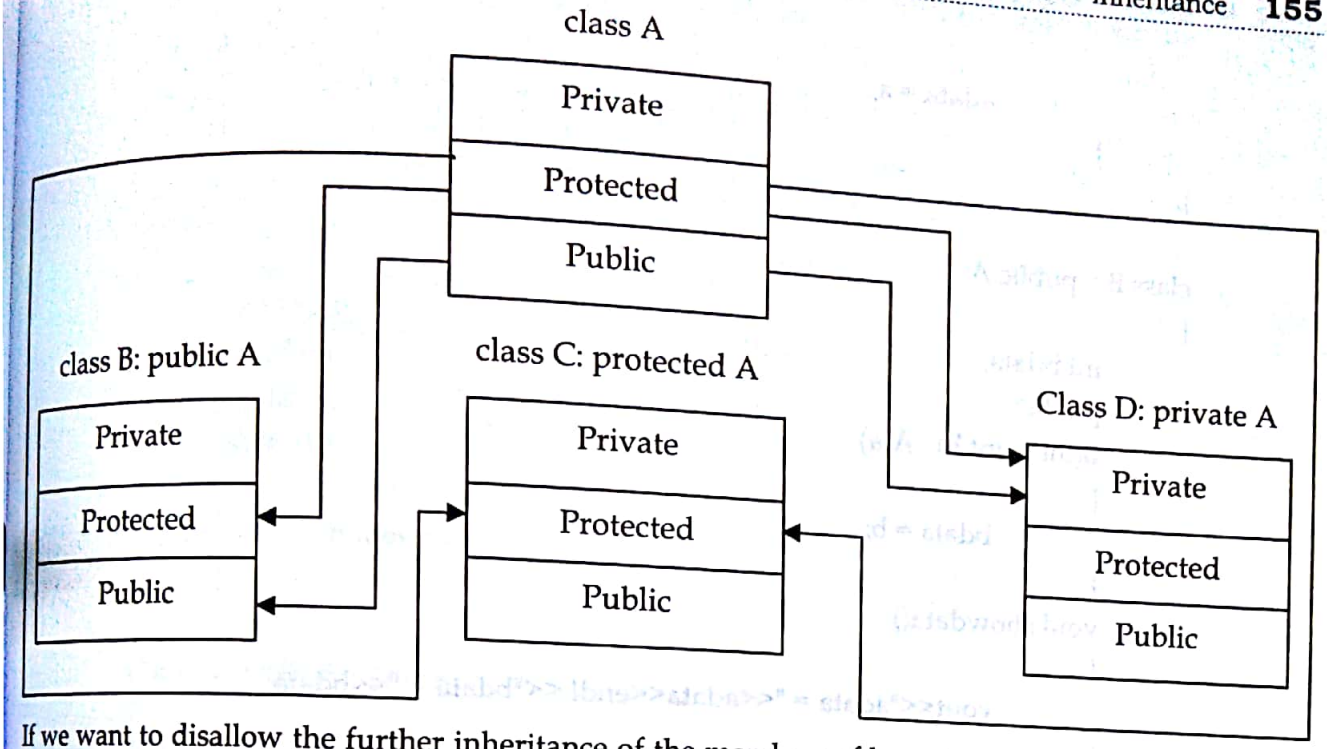
The visibility mode is optional. If present, may be private, protected, or public. The default visibility mode is private. For example,

```
class ABC : [private] [protected] [public] XYZ
{
        members of ABC
};
```

The visibility mode specifies the visibility of inherited members. If the visibility mode is private, public and protected members of the base class become private members in the derived class and therefore these members can only be accessed in the derived class. They are inaccessible from outside of this derived class.

If the visibility mode is protected, both public and protected members of the base class become protected members in the derived class and therefore these members can only be accessed in the derived class and from any class derived from this class. They are inaccessible from outside of these classes. If the visibility mode is public, visibility of public and protected members of the base class do not change. The table below summarizes these concepts:

Base Class Visibility	Derived Class Visibility		
	Pubic Derivation	Private Derivation	Protected Derivation
Private	not visible	not visible	not visible
Protected	Protected	Private	Protected
Public	Public	Private	Protected



If we want to disallow the further inheritance of the members of base class from derived class, then private derivation is used.

8.5 CONSTRUCTORS IN DERIVED CLASSES

As we know, the constructors play an important role in initializing objects. We did not use them earlier in derived classes for the sake of simplicity. As long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

It case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multi-level inheritance, the constructors will be executed in the order of inheritance. Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together when a derived class object is declared. The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class.

EXAMPLE

```
//Derived Class Constructor
#include<iostream.h>
#include<conio.h>
class A
{
        protected:
        int adata;
        public:
        A(int a)
```

```
    {
            adata = a;
    }
};

class B : public A
{
    int bdata;
    public:
    B(int a, int b) : A(a)
    {
            bdata = b;
    }
    void showdata()
    {
            cout<<"adata = "<<adata<<endl <<"bdata = "<<bdata;
    }
};
int main()
{
    B b(5, 6);
    b.showdata();
    getch();
    return 0;
}
```

Output

```
adata = 5
bdata = 6
```

If the base class contains no constructor, we can write the derived class constructor as follows:

```
B(int a, int b)
{
    adata = a;
    bdata = b;
}
```

In case of multiple inheritance, constructors in the base classes are placed in the initializer list in the derived class constructor separated by commas. For example,

```
class A
{
    protected:
    int adata;
    public:

    A(int a)
```

```
    {
        adata = a;
    }
};
class B
{
    protected:
    int bdata;
    public:
    B(int b)
    {
        bdata = b;
    }
};
class C: public A,  public B
{
    int cdata;
    public:
    C(int a, int b, int c) : A(a), B(b)
    {
        cdata = c;
    }
};
```

Order of execution of constructors

The base class constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance. Furthermore, the constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared in the derived class.

EXAMPLE

```
//Order of Execution of Constructors
#include<iostream.h>
#include<conio.h>

class A
{
    public:
    A()
    {
        cout<<"Class A Constructor"<<endl;
    }
};
```

```
class B:public A
{
        public:
        B()
        {
                cout<<"Class B Constructor"<<endl;

        }
        };
        class C: public B
        {
        public:
        C()
        {
                cout<<"Class C Constructor"<<endl;

        }
};
int main()
{
        C x;
        getch();
        return 0;

}
```

Output

```
Class A constructor
Class B constructor
Class C constructor
```

8.6 DESTRUCTORS IN DERIVED CLASSES

If a derived class doesn't explicitly call a constructor for a base class, the default constructor for the parent class, in fact the constructors for the parent classes will be called from the ground up. For example, if you have a base class Vehicle and Car inherits from it, during the construction of a Car, it becomes a Vehicle and then becomes a Car. Destructors for a base class are called automatically in the reverse order of constructors. One thing to think about is with inheritance you may want to make the destructors virtual:

```
class Vehicle
{
        public:
        ~Vehicle();
};
```

*This is important if you ever need to delete a derived class when all you have is a pointer to the base class. Say you have a "Vehicle *v" and you need to delete it. So you call:*

```
delete v;
```

If Vehicle's destructor is non-virtual, the destructor will not call Car::~Car() - just Vehicle::~Vehicle()

Order of execution of Destructors

The derived class destructor is executed first and then the destructor in the base class is executed. In case of multiple inheritances, the derived class destructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the destructors will be executed in the order of inheritance.

EXAMPLE

```
//Destructors under Inheritance
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        ~A()
        {
                cout<<"Class A Destructor"<<endl;
        }
};
class B:public A
{
        public:
        ~B()
        {
                cout<<"Class B Destructor"<<endl;
        }
};
class C: public B
{
        public:
        ~C()
        {
                cout<<"Class C Destructor"<<endl;
        }
};

int main()
{
        {
                C x;
        }//destructor is called at this point
        getch();
        return 0;
}
```

Output

Class C Destructor
Class B Destructor
Class A Destructor

8.7 OVERRIDING MEMBER FUNCTIONS

We can define the functions in derived class having same name and signature as that of base class which is called function overriding. In such situations base class have two versions of same function one derived from based and another defined in the base class itself. And if we call the overridden function by using the object of derived class version of the method defined in derived class is invoked. We can call the version of method derived from base class as below:

obj.classname::method_name;

EXAMPLE

```
//Method Overriding
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        void show()
        {
                cout<<"This is class A";
        }
};
class B : public A
{
        public:
        void show()
        {
                cout<<"This is class B"<<endl;
        }
};
int main()
{
        B b;
        b.show();     //invokes the member function from class B
        b.A :: show(); //invokes the member function from class A
        getche();
        return 0;
}
```

Output

This is class B
This is class A

8.8 AMBIGUITIES IN INHERITANCE

An ambiguity in inheritance arises in two situations: in case of multiple inheritances and in case of hybrid multipath inheritance.

8.8.1 Ambiguity in Multiple Inheritance

Suppose two base classes have an exactly similar member. Also, suppose a class derived from both of these base classes has not this member. Then, if we try to access this member from the objects of the derived class, it will be ambiguous. We can remove this ambiguity by using the syntax

Obj.classname::methodname

EXAMPLE

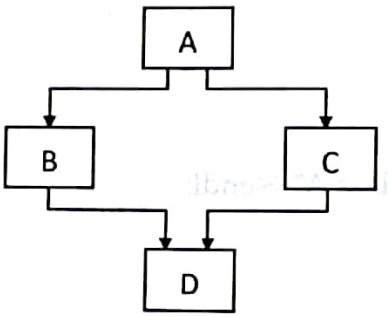
```
//Ambiguity in multiple inheritance and removal of ambiguity
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        void show()
        {
                cout<<"This is class A"<<endl;
        }
};
class B
{
        public:
        void show()
        {
                cout<<"This is class B"<<endl;
        }
};
class C : public A, public B
{

};
int main()
{
        C c;
        c.show(); //ambiguous - will not compile
        c.A :: show(); //OK
        c.B :: show(); //OK
        getche();
        return 0;
}
```

We can also remove this ambiguity by adding a function in class C as follows:

```
class C : public A, public B
{
        public:
        void show()
        {
                A :: show();
                B::show();
        }
};
```

8.8.2 Ambiguity in Multipath Inheritance

Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class as shown in the figure given below, which is called multipath hybrid inheritance. In this case, public or protected member of grandparent is derived in the child class twice which creates confusion to the compiler.

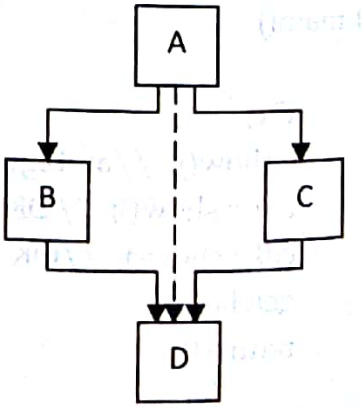


We can remove this kind of ambiguity by using the concept of virtual base classes as described in next section.

8.9 VIRTUAL BASE CLASSES

Consider a situation where we derive a class **D** from two classes **B** and **C** that are each derived from the same class **A**. This creates a diamond-shaped inheritance tree(called hybrid multipath inheritance) and all the public and protected members from class **A** inherited into class **D** twice once through the path A→B→D and again through the path A→B→C. This causes ambiguity and should be avoided. We can remove this kind of ambiguity by using the concept of virtual base class. For this we make direct base classes (B and C) virtual base classes. When this happens, compiler creates direct path from A to D and child class inherits members of grand parent class directly through the broken line. The grandparent is sometimes referred to as indirect base class.

```
class A {
        .....
};
class B : virtual public A {
        .....
};
class C : public virtual A {
        .....
};
class D : public B, public C {
```



.....

};

The keywords virtual and public may be used in either order.

EXAMPLE

```
//Virtual Base Class
#include<iostream.h>
#include<conio.h>
class Person
{
        protected:
        char name[20];
        int age;
        public:
        void getdata()
        {
                cout<<"Enter name and age"<<endl;
                cin>>name>>age;
        }
        void display()
        {
                cout<<"Name::"<<name<<endl;
                cout<<"Age::"<<age<<endl;
        }
};

class Employee: virtual public Person
{
        protected:
        int eid,salary;
        public:
        void getdata()
        {
                cout<<"Enter Employee id and salary"<<endl;
                cin>>eid>>salary;
        }
        void display()
        {
                cout<<"Emp ID::"<<eid<<endl;
                cout<<"Salary::"<<salary<<endl;
        }
};
class Student: public virtual Person
```

```
{
        protected:
        int sid;
        char faculty[20];
        public:
        void getdata()
        {
                cout<<"Enter student ID and faculty"<<endl;
                cin>>sid>>faculty;
        }
        void display()
        {
                cout<<"student ID:"<<sid<<endl;
                cout<<"Faculty::"<<faculty<<endl;
        }
};
class TeachingAssistant: public Employee, public Student
{
        char course[20];
        public:
        void getdata()
        {
                Person::getdata();
                Employee::getdata();
                Student::getdata();
                cout<<"Enter course"<<endl;
                cin>>course;
        }
        void displaydata()
        {
                Person::display();
                Employee::display();
                Student::display();
                cout<<"Course::"<<course;
        }
};
int main()
{
        TeachingAssistant ta;
        ta.getdata();
        cout<<"--------TA Details-------"<<endl;
        cout<<"-------------------------"<<endl;
        ta.displaydata();
```

```
    getch();
    return 0;
}
```

Output

```
Enter name and age
Harry
32
Enter Employee id and salary
908
20000
Enter student ID and faculty
304
Science&Tech
Enter course
Physics
----------TA Details----------
--------------------------------
Name::Harry
Age::32
Emp ID::908
Salary::20000
student ID:304
Faculty::Science&Tech
Course::Physics
```

8.10 AGGREGATION (CONTAINERSHIP)

Inheritance is often called a "kind of" or "is a" relationship. In inheritance, if a class B is derived from a class A, we can say "B is a kind of A". This is because B has all the characteristics of A, and in addition some of its own. For example, we can say that bulldog is a kind of dog: A bulldog has the characteristics shared by all dogs but has some distinctive characteristics of its own.

There is another type of relationship, called a "has a" relationship, or containership. We say that a bulldog has a large head, meaning that each bulldog includes an instance of a large head. In object oriented programming, has a relationship occurs when one object is contained in another. For example,

EXAMPLE

```
//Containership
#include<iostream.h>
#include<conio.h>
class Employee
{
  int eid, sal;
      public:
```

```
        void getdata()
        {
                cout<<"Enter id and salary of employee"<<endl;
                cin>>eid>>sal;
        }
        void display()
        {
                cout<<"Emp ID:"<<eid<<endl<<"Salary:"<<sal;
        }
};
class Company
{
        int cid;
        char cname[20];
        Employee e;
        //Containership, Object of Employee class is included in //Company class
        public:

        void getdata()
        {
                cout<<"Enter id and name of the company:"<<endl;
                cin>>cid>>cname;
                e.getdata();
        }
        void display()
        {
                cout<<"Comp ID:"<<cid<<endl<<"Comp Name:"<<cname;
                e.display();
        }
};
int main()
{
        Company c;
        c.getdata();
        cout<<"###########Company Details################"<<endl;
        c.display();
        getch();
        return 0;
}
```

Output

```
Enter id and name of the company
501
```

```
Citizens
Enter id and salary of an employee
105      30000
###########Company Details###########
Comp ID:501
Comp Name:Cirizens
Emp ID:105
Salary:30000
```

Self Assessment

Fill in the Blanks.

1. Inheritance (or derivation) is the process of creating new classes, called ..... classes, from existing classes, called base classes.
2. Inheritance uses the concept of code ..... Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance.
3. The access specifier or the visibility mode is optional and, if present, may be public, private or protected but by default it is .....
4. When more than one child class is derived from one base class, it is called ..... Inheritance.
5. A protected member can be accessed from within the class where it lies and from any class ..... from this class.
6. If we want to disallow the further inheritance of the members of base class from derived class, then ..... derivation is used.
7. It case of multiple inheritance, the base classes are constructed in the order in which they appear in the ..... of the derived class
8. The ..... class destructor is executed first and then the destructor in the ..... class is executed
9. If we call the overridden function by using the object of derived class, version of the method defined in .....class is invoked.
10. In case of multipath inheritance, public or protected member of grandparent is derived in the child class ..... which creates confusion to the compiler.
11. Inheritance is often called a "kind of" relationship between classes whereas containership is ..... relationship between classes.
12. Virtual destructor is important if you ever need to delete a ..... when all you have is a pointer to the base class.

Exercise

1. Consider a situation where three kinds of inheritance are involved. Explain this situation with an example.
2. What is the difference between protected and private members? Describe with suitable example.
3. Discuss a situation in which the private derivation will be more appropriate as compared to public derivation.

4. Write a C++ program to read and display information about employees and managers.
5. Employee is a class that contains employee number, name, address and department. Manager class and a list of employees working under a manager.
6. Differentiate between public and private inheritances with suitable examples.
7. Explain how a sub-class may inherit from multiple classes.
8. WAP to create class Student having data members name and roll number and necessary member functions. Again create another class Marks having data member's oop, pm, bc, acc, fin and also add necessary functions. Derive the class Result from student and Marks having its own data members total and percentage and member functions for calculating and displaying the data members. Finally create an object of Result class and then read and display its record.
9. What are the ambiguities in inheritance and how do you resolve those ambiguities?
10. What is the purpose of virtual base classes? Explain with suitable example.
11. What will be the output of following code?

```
Class one
{
        public:
        one()
{
        cout<<"Creating class ONE";
        }
        };
        Class two : one
        {
        public:
        two()
        {
        cout<<"Creating class TWO";
        }
};
two myclass = new two();
```

12. How will you make a private member inheritable?
13. How containership is different from inheritance? Explain with nsuitable example.
14. Create class employee with data member's eid, ename, and salary and also add necessary member functions. Create another class company with company name location and 10 employees. Add necessary member functions to the company class. Finally, create an object of company class and then read and display its record along with employee record.

❑❑❑

Chapter 9

Dynamic Polymorphism

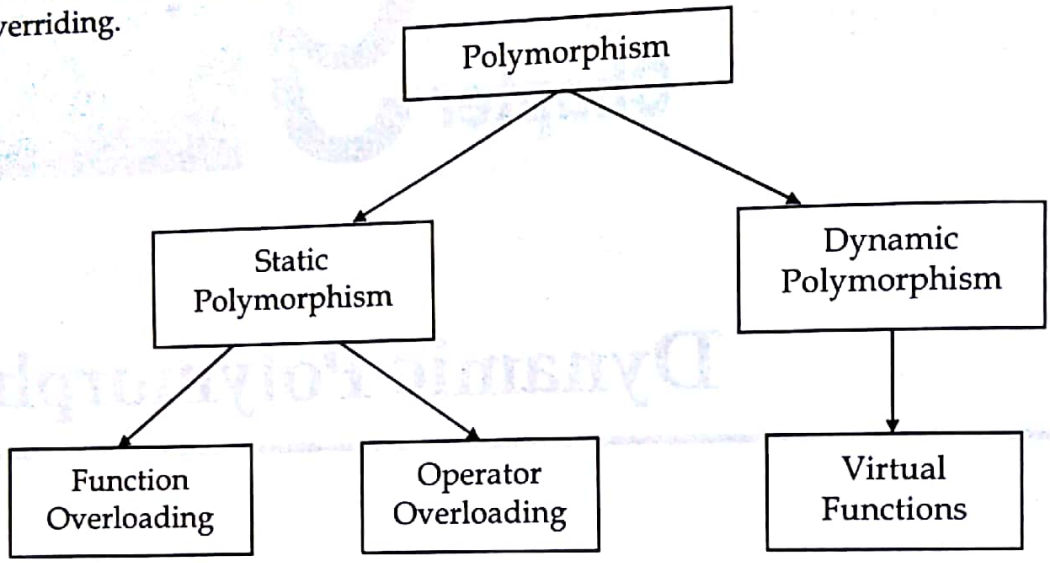
Objectives

After studying this unit, you will be able to:

- Understand base class pointers.
- Demonstrate the virtual functions.
- Recognize the pure virtual functions.
- Describe the abstract classes.
- Explain the dynamic polymorphism.
- Understand virtual destructors.

9.1 INTRODUCTION

Polymorphism means state of having many forms. We have already seen that polymorphism is implemented using the concept of overloaded functions and operators. In this case, polymorphism is called early binding or static binding or static linking. This is also called compile time polymorphism because the compiler knows the information needed to call the function at the compile time and, therefore; compiler is able to select the appropriate function for a particular call at the compile time itself. There is also another kind of polymorphism called run time polymorphism. In this type, the selection of appropriate function is done dynamically at run time. So, this is also called late binding or dynamic binding. C++ supports a mechanism known as virtual functions to achieve run time polymorphism. Run time polymorphism also requires the use of pointers to objects of base class and method overriding.



Role of Polymorphism

1. Same interface could be used for creating methods with different implementations
2. It helps programmers reuse the code and classes once written, tested and implemented. They can be reused in many ways.
3. Polymorphism helps in reducing the coupling between different functionalities
4. Supports building extensible systems

9.2 POINTERS TO BASE CLASSES

We know that an object can be used as its own type or as an object of its base type. In addition, it can be manipulated through an address of the base type. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called up-casting. Thus, pointer to a derived class is type compatible with a pointer to its base class but vice versa is not true. The only limitation of base class pointer is that we can only refer to the members that are defined or declared in base class and not to the members of derived class that are not inherited from base class.

EXAMPLE

```
//Pointers to base class
#include<iostream.h>
#include<conio.h>
class Shape
{
        protected:
        float p,l,b;
```

```
        public:
        void setvalue(int x, int y)
        {
                l=x;
                b=y;
        }
};
class Square:public Shape
{
        public:
        void find_perimeter()
        {
                p=4*l;
                cout<<"Perimeter="<<p<<endl;
        }
};
int main()
{
        Shape bo, *bp;
        Square s;
        bp=&s;
        bp->setvalue(3, 3);
        /*
        bp->find_perimeter();
        it is invalid because  find_perimeter () is not
        inherited from Shape      */
        s.find_perimeter(); //Valid
        getch();
        return 0;
}
```

Output

```
Perimeter=12
```

9.3 VIRTUAL FUNCTIONS

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Furthermore, when we use virtual functions, different functions can be executed by the same function call statement. The information regarding which function to invoke is determined at run time. We can define virtual functions as below:

```
class point
{
        int x;
        int y;
        public:
        virtual void display ( );
};
```

```
void point: : display ( ) //error
{
//Function Body
}
```

The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

9.4 DYNAMIC POLYMORPHISM

We should use virtual functions and pointers to objects of base class to achieve run time polymorphism. For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer. Consider the following code:

EXAMPLE

```
//Dynamic polymorphism
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        virtual void show()
        {
                cout<<"This is class A\n";
        }
};
class B : public A
{
        public:
        void show()
        {
                cout<<"This is class B\n";
        }
};
class C : public A
{
        public:
        void show()
        {
                cout<<"This is class C\n";
        }
};
int main()
{
        A *p, a;
        B b;
```

```
        C c;
        p = &b;
        p->show();
        p = &c;
        p->show();
        p = &a;
        p->show();
        getch();
        return 0;
}
```

Output

```
This is class B
This is class C
This is class A
```

5 NORMAL MEMBER FUNCTIONS VS VIRTUAL FUNCTIONS

we remove the keyword virtual from the class A in above program, all the time show () function of se class is called because at this time function call is made on the basis of type of pointer. member that a base class pointer object can point to any derived objects but the reverse is not true. gain consider the following code without using virtual function:

EXAMPLE

```
//Pointers tro Base class without virtual function
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        void show()
        {
                cout<<"This is class A\n";
        }
};
class B : public A
{
        public:
        void show()
        {
                cout<<"This is class B\n";
        }
};
class C : public A
{
        public:
        void show()
        {
                cout<<"This is class C\n";
        }
```

```
};
int main()
{
        A *p, a;
        B b;
        C c;
        p = &b;
        p->show();
        p = &c;
        p->show();
        p = &a;
        p->show();
        getch();
        return 0;
}
```

Output

```
This is class A
This is class A
This is class A
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

9.6 PURE VIRTUAL FUNCTIONS

Generally a function is declared as virtual inside a base class and we redefine it in the derived classes. The function declared in the base class seldom performs any task. A pure virtual function is one with the expression = 0 added to the declaration – that is, a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Thus, a pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes. We can define pure virtual function as below:

```
class base
{
        private :
        int x, y;
        pubic:
        virtual void getdata ( ) =0;//Pure Virtual Function
        virtual void display ( )=0;
};
class derived : public base
{
        //Data Members
        //Member Functions
};
```

getdata( )=0 doesn't mean assignment of zero to the function. It is only the specification for telling the compiler that the function is pure virtual function.

9.7 ABSTRACT CLASSES AND CONCRETE CLASSES

An abstract class is a class containing at least one pure virtual function. They cannot be instantiated into an object directly. However, we can create pointers and references to an abstract class. Abstract classes are designed to be specifically used as base classes. Only a subclass of an abstract class can be instantiated directly if all inherited pure virtual methods have been implemented by that class.

```
class ABC
{
        Public:
        Fun1( )=0;
        Fun2( )=0;
}
```

With above abstract class, we cannot create it's object of but we can create pointer to ABC class as below:

```
ABC  a; //invalid
ABC *p; //Valid
```

Pure virtual functions are also used where the method declarations are being used to define an interface for which derived classes will supply all implementations. An interface can be constructed with an abstract class having only pure virtual functions, and no data members or ordinary methods.

A concrete class is a class that can be used to create an object. It has no pure virtual functions. **Concrete class** can be used to create an object. You must extend an abstract class and make a concrete class to be able to then create an object.

EXAMPLE

```
//Abstract base classes and pure virtual functions
#include <iostream.h>
#include<conio.h>
class Polygon //abstract class
{
        protected:
        int width, height;
        public:
        void setvalues (int a, int b)
        {
                width=a; height=b;
        }
        virtual int area (void) =0;
};
class Rectangle: public Polygon //concrete class
{
        public:
        int area (void)
        {
                return (width * height);
        }
};
class Triangle: public Polygon //concrete class
```

```
{
        public:
        int area (void)
        {
                return (width * height / 2); }

};
int main ()
{
        Rectangle rect;
        Triangle trgl;
        Polygon * poly = &rect;
        poly->setvalues(4,5);
        cout<<"Area of Rectangle="<<poly->area()<< endl;
        poly=&trgl;
        poly->setvalues(6,5);
        cout<<"Area of Traingle="<<poly->area()<< endl;
        getch();
        return 0;
}
```

Output

```
Area of Rectangle=20
Area of Traingle=15
```

Remember that we cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion.

9.8 VIRTUAL DESTRUCTORS

Once an inheritance hierarchy is created, with memory allocations occurring at each stage in the hierarchy, it is necessary to be very careful about how objects are destroyed so that any memory leaks are avoided. In order to achieve this, we make use of a virtual destructor. A virtual destructor ensures that when objects of derived subclasses go out of scope or are deleted the order of destruction of each class in a hierarchy is carried out correctly. If the destruction order of the class objects is incorrect, it can lead to what is known as a *memory leak*. This is undesirable behavior as the operating system has no mechanism to regain the lost memory. Since memory is a finite resource, if this leak persists over continued program usage, eventually there will be no available RAM space to carry out other programs. Virtual destructors are useful when you delete an *instance* of a derived class through a pointer to base class. When we use the concept of virtual destructors, it forces the compiler to call the destructors of derived class while using base pointers with delete operator. Example given below explains the behavior of virtual vs. non-virtual destructor.

EXAMPLE

```
//Virtual Destructors
#include<iostream.h>
#include<conio.h>
class Base1
{
        public:
        ~Base1() { cout << "~Base1()\n"; }
};
```

```
class Derived1 : public Base1
{
        public:
        ~Derived1() { cout << "~Derived1()\n"; }
};
class Base2
{
        public:
        virtual ~Base2() { cout << "~Base2()\n"; }
};
class Derived2 : public Base2
{
        public:
        ~Derived2() { cout << "~Derived2()\n"; }
};
int main()
{
        Base1* bp = new Derived1;
        delete bp;
        Base2* b2p = new Derived2;
        delete b2p;
        getch();
        return 0;
}
```

Output

```
~Base1()
~Derived2()
~Base2()
```

Then you run the program, you'll see that *delete bp* only calls the base-class destructor, while *delete b2p* calls the derived-class destructor followed by the base-class destructor, which is the behavior we desire. Forgetting to make a destructor virtual is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak.

Self Assessment

Fill in the Blanks.

1. C++ supports a mechanism known as ..... to achieve run time polymorphism.
2. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called .....
3. When we use virtual functions, different functions can be executed by the same function call statement. The information regarding which function to invoke is determined at ..... time.
4. We should use virtual functions and pointers to ..... to achieve run time polymorphism.
5. A virtual function cannot be a ..... member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.
6. In case of pure virtual function, the compiler requires each derived class to either define the function or ..... it as a pure virtual function.

7. An ..... class is a class containing at least one pure virtual function as its member.
8. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during ..... operator.

Exercise

1. Make the distinction between virtual functions and virtual base class? Describe their use with suitable example.
2. What is meant by upcasting? Describe use of upcasting with any example of your choice.
3. Is method overriding necessary for achieving? Justify this in your own language with support of suitable example.
4. What is the difference between virtual and non-virtual member functions?
5. How can a member function in my derived class call the same function from its base class?
6. Write a program to use constructors and destructors in inheritance.
7. Debug the following program:

```
class one
{
        int a;
        public:
        void set(int a)
        {
                a = a;
        }
void show()
{
        cout << a;
}
};
main()
{
        one O1, O2;
        O1.set(10);
        O2.show();
}
```

8. What is the purpose of a pure virtual function? Illustrate with a suitable example.
9. Create a class Teachmedia with data member's publisher author and price, and with member functions for reading and displaying data members. Derive the class Book from Teachmedia and add its own data member np(number of pages). Also add member functions for reading and displaying data members. Again derive another class CD from Teachmedia and add its own data member size. Also add member functions necessary. Finally create objects of Book and CD class and then read and display their records by using parent/base class pointer.
10. Create a class Shape with data members length, breadth and height and with member function Area (pure virtual). Derive the class Rectangle from Shape and override Area function. Again derive another class Triangle and override the member function Area. Finally create objects of Rectangle and Triangle class and calculate their areas by using base class pointer.

Chapter 10

Exception Handling

Objectives

After studying this unit, you will be able to:

- Differentiate errors from exceptions.
- Identify necessity of exception handling.
- Learn exception handling mechanism with try....catch.
- Use finally for cleanup operation.
- Creating and throwing user defined exceptions.

10.1 INTRODUCTION

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. When exception is occurred it causes the abnormal termination of program. To avoid abnormal termination of programs in case of exceptional circumstances and to continue normal flow of program execution exceptional handling mechanism must be used. During the execution of a program, the computer will face two types of situations: those it is prepared to deal with and those it doesn't like (exception). Imagine you write a program that asks the user to supply two numbers to perform a calculation. Here is such a program:

EXAMPLE 1

```
//Simple program to multiply two numbers
#include <iostream.h>
#include<conio.h>
int main()
{
        double a, b, c;
        cout << "Please provide two numbers\n";
        cout << "First Number: ";
        cin >> a;
        cout << "Second Number: ";
        cin >> b;
        c = a * b;
        cout << "\n" << a << " * " << b << " = " << c << "\n\n";
        getch();
        return 0;
}
```

Output

```
Please provide two numbers
First Number: 3
Second Number: 5
a*b=15
```

This is a classic easy program. When it comes up, the user is asked to simply type two numbers; the program would use them to perform a multiplication and display the result. Imagine that a user that decides to type string values for the requested values. Since a program such as this one is not prepared to multiply two strings, it would not know what to do. The only alternative the compiler would have is to send the problem to the operating system, hoping that the OS would know what to do. What actually happens is that, whenever the compiler is handed a task, it would try to perform the assignment. If it can't perform the assignment, it would throw an error. As a programmer, if you can anticipate the type of error that could occur in your program, you can catch the error yourself and deal with it by telling the compiler what to do when this type of error occurs.

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing. Exceptions might include conditions such as division by zero, access to an array outside of its bounds, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc. When a program encounters an exceptional condition, it is

important that it is identified and dealt with effectively. C++ provides built-in language features to detect and handle exceptions. The purpose of the exception handling mechanism is to provide means to detect and report an exceptional circumstance so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

- Find the problem (**Hit the exception**).
- Inform that an error has occurred (**Throw the exception**).
- Receive the error information (**Catch the exception**).
- Take corrective action (**Handle the exception**).

The error handling code basically consists to two segments: one to detect errors and to throw exceptions and the other to catch the exceptions and to take appropriate actions. Exception handling explained in this chapter is a new feature introduced by ANSI-C++ standard. If you use a C++ compiler that is not adapted to this standard it is possible that you cannot use this feature. Following are main advantages or uses of exception handling over traditional error handling.

- **Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
- **Methods can handle any exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)
- **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

10.2 EXCEPTION HANDLING MECHANISM: try, catch , and throw

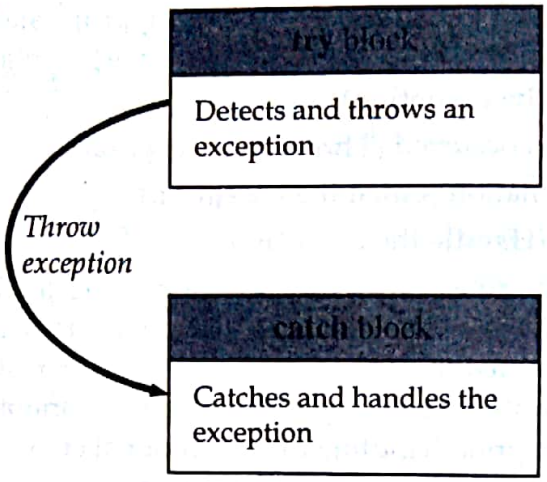
Exception handling mechanism in C++ is basically built upon three keywords: **try**, **throw**, and **catch**. The keyword **try** is used to surround a block of statements, which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using the **throw** statement situated either in the try block or in functions that are invoked from within the try block. This is called throwing an exception and the point at which the **throw** is executed is called the throw point.

A catch block defined by the keyword **catch** catches the exception thrown by the **throw** statement and handles it appropriately. This block is also called exception handler. The catch block that catches an exception must immediately follow the try block that throws an exception. The figure below shows the exception handling mechanism if try block throws an exception.

General form of using try catch statement is the following:

```
try
{
    // code to be tried
    throw exception;
}
catch (type exception)
{
```

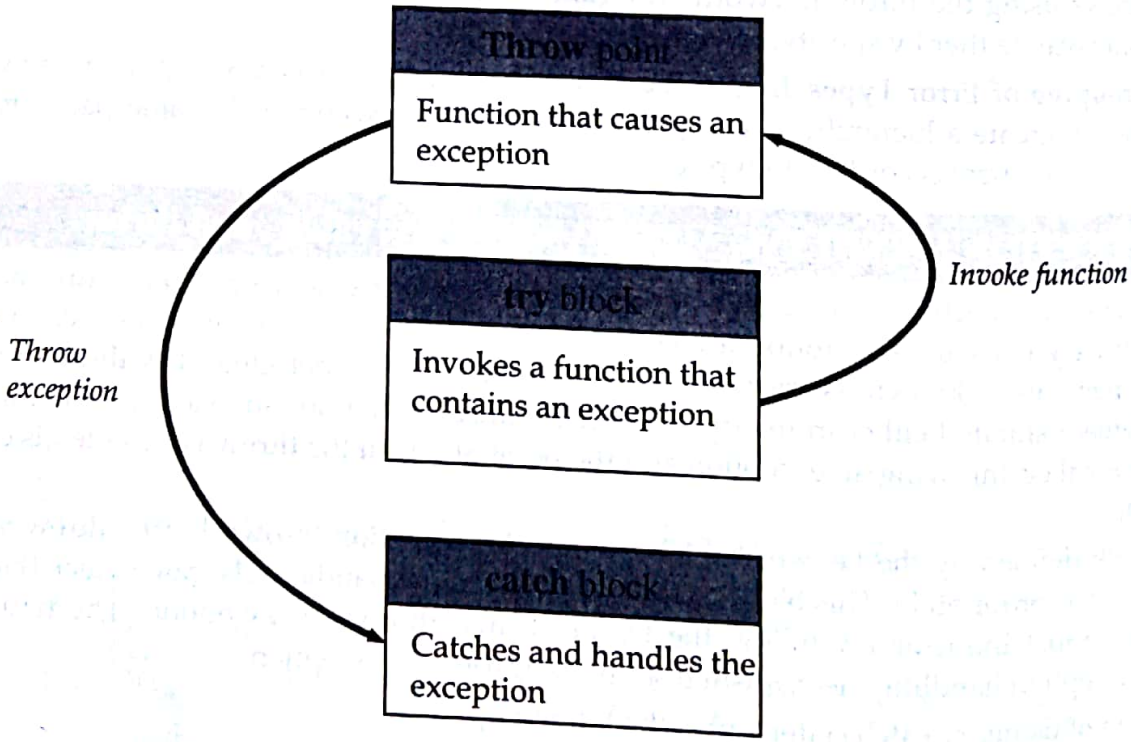
```
        // code to be executed in case of exception
    }
```



Operation of try……catch blocks:

- The code within the **try** block is executed normally. In case that an exception takes place, this code must use the **throw** keyword and a parameter to throw an exception. The type of the parameter details the exception and can be of any valid type.
- If an exception has taken place, that is to say, if it has executed a **throw** instruction within the **try** block, the **catch** block is executed receiving as parameter the exception passed by **throw**.

The figure below shows the exception handling mechanism if function invoked by try block throws an exception.



EXAMPLE

```
//Try block throwing exception
#include<iostream.h>
#include<conio.h>
int main()
{
```

```
int a;
int b;
cout<<"Enter values of a & b:\n";
cin>>a>>b;
try
{
        if(b == 0)
                throw b;
        else
                cout<<"Result = "<<(float)a/b;
}
catch(int i)
{
        cout<<"Divide by zero exception: b = "<<i;
}
cout<<"\nEND";
getch();
return 0;
}
```

Output

First Execution

```
Enter values of a & b:
9       4
Result=2.25
END
```

Second Execution

```
Enter values of a & b:
7       0
Divide by zero exception: b=0
END
```

If a function throws an exception but it does not uses try.....catch block for exception handling, we must enclose function call statement in try block and exception thrown by the function must be handled inside the caller of the function.

EXAMPLE

```
//Function invoked by try block throwing exception
#include<iostream.h>
#include<conio.h>
void divide(int a, int b)
{
        if(b == 0)
        throw b;
        else
        cout<<"Result = "<<(float)a/b;
}
```

```
int main()
{
        int a, b;
        cout<<"Enter values of a & b:\n";
        cin>>a>>b;
        try
        {
                divide(a, b);
        }
        catch(int i)
        {
                cout<<"Divide by zero exception: b = "<<i;
        }
        cout<<"\nEND";
        getche();
        return 0;
}
```

Output

First Run

```
Enter the values of a & b:
5
2
Result = 2.5
END
```

Second Run

```
Enter the values of a & b:
9
0
Divide by zero exception: b = 0
END
```

In the first example, if exception occurs in the try block, it is thrown and the program control leaves in the try block and enters the catch block. In the second example, try block invokes the function divide. If exception occurs in this function, it is thrown and control leaves this function and enters the catch block in calling function (i.e. main function).

Note that, exceptions are used to transmit information about the problem. If the type of exception thrown matches the argument type in the **catch** statement, then only catch block is executed for handling the exception. After that, control goes to the statement immediately after the catch block. If they do not match, the program is aborted with the help of **abort ()** function, which is invoked by default. In this case, statements following the catch block are not executed. When no exception is detected and thrown, the control goes to the statement immediately after the catch block skipping the catch block.

10.3 THROW STATEMENT IN DETAIL

When an exception is detected, it is thrown using the **throw** statement in one of the following forms:

```
throw(exception);
throw; //used for rethrowing an exception
```

The operand exception may be of any type (built-in and user-defined), including constants. When an exception is thrown, the catch statement associated with the try block will catch it. That is, the control exits the current try block, and is transferred to the catch block after the try block. Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement. A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block. For example:

EXAMPLE

```
//Using throw statement
#include<iostream.h>
#include<conio.h>
void divide(int a, int b)
{
try
{
        if(b == 0)
                throw b;
        else
                cout<<"Result = "<<(float)a/b;
 }
 catch(int)
 {
        throw;
 }
}
int main()
{
        int a, b;
        cout<<"Enter values of a & b:\n";
        cin>>a>>b;
        try
        {
                divide(a, b);
        }
        catch(int i)
        {
                cout<<"Divide by zero exception: b = "<<i;
```

```
        }
        cout<<"\nEND";
        getch();
        return 0;
}
```

Output

First Run

```
Enter the values of a & b:
10
4
Result = 2.5
END
```

Second Run

```
Enter the values of a & b:
10
0
Divide by zero exception: b = 0
END
```

10.4 CATCH STATEMENT (EXCEPTIONS WITH AND WITHOUT ARGUMENTS)

Code for handling exceptions is included in catch blocks. The general form of catch block is:

```
catch(type arg)
{
        Body of catch block
}
```

The type indicates the type of exception that catch block handles. The parameter arg is optional. If it is named, it can be used in the exception handling code. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed. After its execution, the control goes to the statement immediately following the catch block. If an exception is not caught, abnormal program termination will occur. The catch block is simply skipped if the catch statement does not catch an exception,

10.4.1 Multiple Catch Statements

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one **catch** statement with a **try** as shown below:

```
try
{
Try block
}

catch(type1 arg)
{
        Catch block1
}
```

```
catch(type2 arg)
{
        Catch block 2
}
.....
.....
}
catch(typeN arg)
{
        Catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After that, the control goes to the first statement after the last **catch** block for that **try** skipping other exception handlers. When no match is found, the program is terminated.

EXAMPLE

```
//Program using multiple catch blocks for a single try block
#include <iostream.h>
#include<conio.h>
int main()
{
        int size;
        cout<<"Enter Size"<<endl;
        cin>>size;
        try
        {
                char * mystring;
                if(size<=0)
                throw 's';
                mystring = new char [size];
                for (int n=0; n<=100; n++)
                {
                        if (n>size-1)
                        throw n;
                }
        }
        catch (int i)
        {
                cout << "Exception: ";
                cout << "index " << i << " is out of range" << endl;
        }
        catch (char c)
        {
```

```
            cout << "Exception: "<<"Size must be non-zero positive number:"<< endl;
        }
        getch();
        return 0;
    }
```

Output

First Run
Enter Size:
10
Exception: Index 10 is out of range
Second Run
Enter Size:
-2
Exception: Size must be non-zero positive number

10.4.2 Catching All Exceptions

We can also define a **catch** block that captures all the exceptions independently of the type used in the call to **throw**. For that we have to write three points instead of the parameter type and name accepted by **catch**:

```
try
{
    // code here
}
catch (...)
{
    cout << "Exception occurred";
}
```

EXAMPLE

```
//Showing how to catch all exception by using single catch block
#include <iostream.h>
#include<conio.h>
int main()
{
    double operand1, operand2, result;
    char opr; //operator
    cout << "To proceed, enter two numbers\n";
    try
    {
        cout << "First Number: ";
        cin >> operand1;
        cout << "Operator:";
        cin >> opr;
        cout << "Second Number: ";
```

```
            cin >> operand2;
            // Make sure the user typed a valid operator
            if(opr!='+' && opr!='-' && opr!='*' && opr!='/')
                  throw opr;
            // Find out if the denominator is 0
            if(opr == '/')
                  if(operand2== 0)
                        throw 0;
            // Perform an operation based on the user's choice
            switch(opr)
            {
                  case '+':
                        result = operand1 + operand2;
                        break;
                  case '-':
                        result = operand1 - operand2;
                        break;
                  case '*':
                        result = operand1 * operand2;
                        break;
                  case '/':
                        result = operand1 / operand2;
                        break;
            }
            // Display the result of the operation
      cout<<operand1 << opr << operand2 << " = "<<result<<"\n";
      }
      catch(...)
      {
            cout << "Exception is occurred!!!!\n";
      }
      getch();
      return 0;
}
```

Output

First Run

```
To Proceed Enter two numbers
First Number: 10
Operator: +
Second Number: 7
10+7=17
```

Second Run

```
To proceed Enter two numbers
First Number: 15
Operator: &
Second Number: 6
Exception is occurred!!!!
```

Third Run

```
To Proceed Enter two numbers
First Number: 8
Operator: /
Second Number:0
Exception is occurred!!!!
```

10.5 NESTED TRY-CATCH

C++ allows you to nest exceptions; using the same techniques we applied to nest conditional statements. This means that you can write an exception that depends on, and is subject to, another exception. To nest an exception, write a try block in the body of the parent exception. The nested try block must be followed by its own catch(es). To effectively handle the exception, make sure you include an appropriate throw in the try block. The general form of nested try catch is:

```
try
{
        try
        {
        // code here
        }
        catch ( )
        {
        //Exception handler
        }
}
catch ( )
{
        //Handler
}
```

EXAMPLE

```
//Showing use of netsed try.....catch() statement
#include <iostream.h>
#include<conio.h>
#include <string.h>
int main()
{
        double Operand1, Operand2, Result;
        char opr;
```

```
try
{
        cout << "To proceed, enter\n";
        cout << "First Number: ";
        cin >> Number1;
        cout << "An Operator: ";
        cin >> opr;
        cout << "Second Number: ";
        cin >> Number2;

        if(opr!='+' && opr!='-' && opr!='*' && opr!='/')
                throw opr;
                switch(opr)
                {
                case '+':
                        Result = Operand1 + Operand2;
                        cout << "\n" << Operand1 << " + "  << Operand2 << " = "
                        << Result;
                        break;
                case '-':
                        Result = Operand1 - Operand2;
                        cout << "\n" << Operand1 << " - " << Operand2 << " = "
                        << Result;
                        break;
                case '*':
                        Result = Operand1 * Operand2;
                        cout << "\n" << Operand1 << " * " << Operand2 << " = "
                        << Result;
                        break;
                case '/':
                /*The following exception is nested in the previous try*/
                        try {
                                if(Operand2 == 0)
                                throw "Division by 0 not allowed";
                                Result = Operand1 / Operand2;
                                cout<<"\n"<< Operand1 << " / " << Operand2
                                << " = " << Result;
                        }
                        catch(const  char * Str)
                        {
                                cout << "\nBad Operation:"<< Str;
                        }
                        break;
                }
```

```
        }

        catch(const char n)
        {
                cout << "\nOperation Error: " << n << " is not a valid operator";
        }
        cout << "\n\n";
        getch();
        return 0;
}
```

Output

First Run

```
Enter to Proceed:
First Number: 20
An Operator: *
Second Number: 9
20*9=180
```

Second Run

```
Enter to Proceed:
First Number: 15
Operator: $
Second Number: 6
Operation Error: $ in not a valid operator
```

Third Run

```
First Number: 8
Operator: /
Second Number:0
Bad Operation: Division by zero is not allowed
```

10.6 SPECIFYING EXCEPTIONS

It is also possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form is as follows:

```
type function(arg-list) throw (type-list)
{
        Function body
}
```

The type-list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty. Hence the specification in this case will be

```
type function(arg-list) throw ()
{
```

```
    Function body
}
```

A function can only be restricted in what types of exception it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function (and not within the function).

EXAMPLE

```
//Program to restrict the exception that can be thrown from a function
#include<iostream.h>
#include<conio.h>
void test(int x) throw (int, double)
{
  if(x == 0)
    throw x;
  if(x == 1)
    throw 1.0;
  if(x == 2)
    throw 'a';
}
int main()
{
      int n;
      cout<<"Enter a number{0,1,2}"<<endl;
      cin>>n;
  try
  {
    test(n);
  }
  catch(int m)
  {
    cout<<"Caught an integer\n";
  }
  catch (double d)
  {
        cout<<"Caught a double";
  }
  catch (char c)
  {
        cout<<"Caught a character";
  }
      getch();
      return 0;
}
```

Output:

First Run
Enter a number{0,1,2}: 0
Caught an integer
Second Run
Enter a number{0,1,2}: 1
Caught a double
Third Run
Enter a number{0,1,2}: 2
Caught an character

10.7 STANDARD EXCEPTIONS

The C++ standard library provides a base class specifically designed to declare objects to be thrown as exceptions. To make use of the standard exceptions we have to include the exception header file. All of the exceptions thrown by parts of the C++ Standard library will throw exceptions derived from the std::exception class. These exceptions are:

- → **bad_alloc:** A bad_alloc is thrown by new if an allocation failure occurs.
- → **bad_cast:** A bad_cast is thrown by dynamic_cast when it fails with a referenced type.
- → **bad_exception:** A bad_exception is thrown when an exception type doesn't match any catch
- → **bad_typeid:** A bad_typeid is thrown by typeid
- → **ios_base::failure:** An ios_base::failure is thrown by functions in the iostream library.

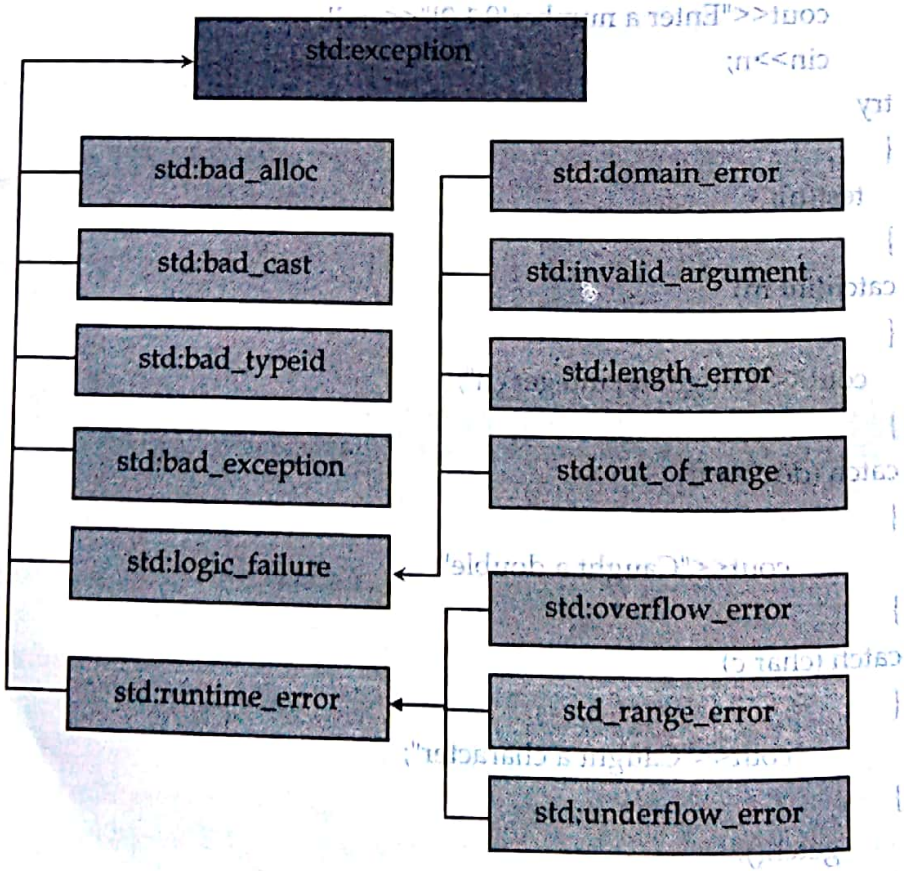


Figure 10.1: Hierarchy of standard exceptions.

So let's use it in an example, but be warned this example can lock your computer! It should not happen, but it might.

EXAMPLE

```
#include<iostream>
#include <exception>
int main()
{
    try
    {
        int * my_array1= new int[100000000];
        int * my_array2= new int[100000000];
        //int * my_array3= new int[100000000];
        //int * my_array4= new int[100000000];
        //add more if needed.
    }
    catch (bad_alloc&)
    {
        cout << "Error allocating the requested memory." << endl;
    }
    return 0;
}
```

You should add more my_array's until exception occurs. It is recommended to use try blocks if you use dynamic memory allocation. This way you are in control if an allocation fails.

Self Assessment

Fill in the Blanks.

1. Exceptions are ..... or unusual conditions that a program may encounter while executing.
2. The error handling code basically consists to two segments, one to detect errors and to throw exceptions, and the other to ..... the exceptions and to take appropriate actions.
3. The keyword .....is used to surround a block of statements, which may generate exceptions.
4. The catch block that catches an exception must immediately follow the .....block that throws an exception.
5. If the type of exception thrown matches the ..... type in the catch statement, then only catch block is executed for handling the exception.
6. A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke .....without any arguments
7. If an exception is not caught by any catch block, abnormal program ..... will occur.
8. When an exception is thrown, the exception handlers are searched in order for an appropriate match. The ..... handler that yields a match is executed.

9. We can also define a catch block that captures all the exceptions independently of the type used in the call to throw. For that we have to write ..... instead of the parameter type and name accepted by catch.
10. It is also possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a ..... list clause to the function definition.

Exercise

1. Differentiate between exceptions and errors. Give at least three examples of exceptional conditions.
2. Why it is necessary to handle the exceptions? Explain exception handling mechanism in suitable example.
3. What happens when a try block is followed by multiple catch blocks? Explain the operation with proper example.
4. In which case it necessary to specify multiple catch blocks after a try block? Explain the situation with example.
5. How do you handle the exception when exception handling code is not written in the function?
6. Is it possible to nest try.....catch() statement? Show an example of nested try.....catch statement.
7. Write a program that throws and handles exceptions in following situations:
   a. Division by zero
   b. Array index out of bound
   c. Finding square root of negative numbers
   d. Trying arithmetic calculation with non-numerical data
8. How do you catch all exceptions by using single catch blocks? Modify above program to handle all exceptions by using a single catch block.
9. How the number of exception thrown from a function can be restricted? What happens if the function throws an unspecified exception?
10. When an exception is rethrown from a function? Explain the exception rethrowing mechanism with example.

❑❑❑

Chapter 11

Templates

Objectives

After studying this unit, you will be able to:

- Understand important of templates.
- Use function templates.
- Use class templates.
- Describe template specialization.

11.1 INTRODUCTION

Many C++ programs use common data structures like stack, queues and lists. A program may require a queue of customers and queue of messages. You could easily implement a queue of customers, then take the existing code and implement the queue of messages. The program grows and now there is need of queue of orders. So just take the queue of messages and convert that to the queue of orders (copy, paste, find, replace???????). Every time you need to make changes to the implementation of queue which is not so easy. Reinventing source code is not an intelligent approach in an object oriented environment which encourages re-usability. It seems to make more sense to implement a queue that contains any arbitrary type rather than duplicating code.

C++ templates allow us to implement a generic queue template that has a parameter type T. That can be replaced with any actual type (Basic or Object) by compilers. C++ provides two kinds of templates: *class templates and function templates*. Function templates are used to write generic functions that can be used with arbitrary types and class templates are used to define generic containers. Template can be instantiated for all data types, eliminating duplication of the source code. The identifier, 'type', can be used within the function body of a function that, otherwise, remains unchanged.

11.2 FUNCTION TEMPLATES

In C ++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. In the example of the max ( ) function, which returns the greater of the two values passed to it this function would have to be coded for every data type being used. Thus, you will end up coding the same function for each of the types, like int, float, char, and double. A few versions of max ( ) are:

```
int max ( int x, int y)
{
        return x > y ? x : y
}
char max ( char x, char y )
{
        return x > y ? > x : y
}
double max (double x , double y)
{
        return x > y ? x : y
}
float max ( float x, float y)
{
        return x > y ? x : y
}
```

Here you can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided using the template utility provided by C ++. A template function may be defined as an unbounded function in which all the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary. Template functions are defined by using the keyword, template. Templates are blueprints of a function that can be applied to different data types.

Syntax of Template

```
template < class type 1, type 2 ... >
void function - name ( type 2 parameter 1, type 1 parameter 2 )
{
        //Function Body
}
```

Example:

```
Template < class X >
X min ( X a , X b )
{
        return (a < b ) ? a : b ;
}
```

This list of parameter types is called the formal parameter list of the template, and it cannot be empty. Each formal parameter consists of the keyword, type name, followed by an identifier. The identifier can be built-in or user-defined data type, or the identifier type. When the function is invoked with actual parameters, the identifier type is substituted with the actual type of the parameter. This allows the use of any data type. The template declaration immediately precedes the definition of the function for which the template is being defined. The template declaration, followed by the function definition, constitutes the template definition. The template of the max ( ) function is coded below:

EXAMPLE

```
//Finding maximum of two values by using template function
#include <iostream.h>
#include<conio.h>
template <class T>
T GetMax (T a, T b)
{
        T result;
        result = (a>b)? a : b;
        return (result);
}
int main ()
{
        int a=5, b=6, k;
        float l=10, m=5, n;
        char x='a', y='b', z;
        k=GetMax(a,b);
        n=GetMax(l,m);
        z=GetMax(x,y);
        cout <<"Larger of ingeghers::"<<k<< endl;
        cout <<"Larger of floats::"<<n<< endl;
        cout <<"Larger of characters::"<<z<< endl;
        getch();
        return 0;
```

```
}
```

Output

```
Larger of ingeghers::6
Larger of floats::10
Larger of characters::b
```

In the example, the list of parameters is composed of only one parameter. The next line specifies that the function takes two arguments and returns a value, all of the defined in the formal parameter list. See what happens if the following command is issued, keeping in mind the template definition for max ( ) :

```
max ( a , b ) ;
```

in which a and b are integer type variables. When the user invokes max ( ) , using two int values, the identifier, 'T', is substituted with int, wherever it is present. Now max ( ) works just like the function int max (int, int) defined earlier, to compare two int values. Similarly, if max ( ) is invoked using two double values, 'T' is replaced with 'double'. This process of substitution, depending on the parameter type passed to the function, is called template instantiation. The template specifies how individual functions will be constructed, given a set of actual types. The template facility allows the creation of a blueprint for a function like max ( ).

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b)
{
        return (a<b?a:b);
}
```

In this case, function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int x,y;
long z;
x = GetMin (y,z);
```

even though y and z have different types, it works since the compiler can determine the appropriate instantiation anyway.

EXAMPLE

```
//Finding miminum of two values of different types by using template function
#include <iostream.h>
```

```
#include<conio.h>
template <class T,class U>
T GetMin (T a, U b)
{
        T result;
        result = (a<b)? a : b;
        return (result);
}
int main ()
{
        int x=5,r;
        long y=9;
        r=GetMin(x,y);
        cout <<"Smaller of two numbers is::"<<r<< endl;
        getch();
        return 0;
}
```

Output

```
        Smaller of two numbers is::5
```

11.3 CLASS TEMPLATES

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates. One good area where this C++ Class Templates are suited can be container classes. Very famous examples for these container classes will be the STL classes like vector, list etc., Once code is written as a C++ class template, it can support all data types. Though very useful, It is advisable to write a class as a template after getting a good hands-on experience on the logic (by writing the code with normal data types). Suppose that you are writing a C++ program that requires two stacks--one for integer data and one for string data. You could implement these classes as follows:

```
//Implementing stack of integers
class IntStack
{
        int top;
        int st[100];
        public:
        IntStack()
        {
                top = -1;
        }
        void push(int i)
        {
                st[++top] = i;
        }
```

```
        int pop()
        {
                return st[top--];
        }
};

//Implementing stack of strings
class StringStack
{
public:
        StringStack()
{
top = -1;
}
        void push(string i)
        {
st[++top] = i;
}
        string pop()
        {
return st[top--];
}
};
```

Notice that the only difference between these classes (other than the name of the class) is the type of data that's put onto the stack. C++ allows you to define a single, template class to represent a stack of any possible datatype (including a user-defined datatype).

Syntax

```
Template < class type 1, ... >
Class class - name
{
        public
        type 1 var,
        ...
};
```

```
//The declaration of class Template for implementing stack would look like this:
        template <class T>
        class Stack
        {
                int top;
        T st[100];
        public:
                Stack()
        {
```

```
top = -1;
}
        void push(T i)
        {
st[++top] = i;
}
        T pop()
        {
        return st[top--];
}
};
```

The T represents the type of stack desired. The main program would declare and use the stacks as follows:

```
Stack<int> ii;
Stack<string> ss;
ii.push(25);
ss.push("Hello");
```

EXAMPLE

```
//Class templates to find larger of two numbers
#include <iostream.h>
#include<conio.h>
template <class T>
class mypair
{
T a, b;
public:
mypair (T first, T second)
{
        a=first; b=second;
}
T getmax ()
{
        T retval;
        retval = a>b? a : b;
        return retval;
}
};
int main ( )
{
  mypair<int> object1(100, 75);
  cout <<"Larger="<<object1.getmax()<<endl;
  mypair<double> object2(90.80, 98.56);
```

```
    cout <<"Larger="<<object2.getmax()<<endl;
    getch();
    return 0;
}
```

Output

```
Larger=100
Larger=98.56
```

11.4 TEMPLATE AND INHERITANCE

It is possible to inherit from a template class. All the usual rules for inheritance and polymorphism apply. If we want the new, derived class to be generic it should also be a template class; and pass its template parameter along to the base class.

EXAMPLE

```
//program to demonstrate inheritance in template classes
#include<iostream.h>
Template<class T>
Class Base
{
        private:
                T a;
        public:
                void set(T &val)
                {
                        a=val;
                }
                void display()
                {
                        cout<<"a="<<a<<endl;
                }
};
template<class T>
class Derived: public Base <T>
{
        private:
                int b;
        public:
                void se(T &val1, T &val2)
                {
                        Base<T>:set(val1);
                        b= val2;
                }
                void display()
                {
```

```
                    Base<T>:display();
                    cout<<"a="<<a<<endl;
                }
}
void main()
{
        Derived<int> x;
        x.set(100, 75);
        x.display();
}
```

Output

```
a=100
b=74
```

11.5 CLASS TEMPLATE SPECIALIZATION

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template. For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char, it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type as below:

EXAMPLE

```
//Class template specialization
#include <iostream.h>
#include<conio.h>
template <class T>
class mycontainer
{
        T element;
        public:
        mycontainer (T arg)
        {
                element=arg;
        }
        T increase ()
        {
                return ++element;
        }
};
// class template specialization
template<>
```

```
class mycontainer<char>
{
        char element;
        public:
        mycontainer<char>(char arg)
        {
                element=arg;
        }
        char uppercase()
        {
                if((element>='a')&&(element<='z'))
                element+='A'-'a';
                return element;
        }
};
int main ()
{
        mycontainer<int> myint (7);
        mycontainer<char> mychar('b');
        cout <<"After Increment:"<<myint.increase()<< endl;
        cout <<"Upeer Case is:"<<mychar.uppercase()<< endl;
        getch();
        return 0;
}
```

Output

```
After Increment: 8
Upper Case is: B
```

11.6 FUNCTION TEMPLATE SPECIALIZATION

In example 1, If the function template is instantiated with a template argument of type String, the generated instance fails to compile because the String class does not support either the less-than or greater-than operators. The code given below shows how we can specialize a function template. Remember that the general function template must have been declared before providing function template specialization.

EXAMPLE

```
//Function Template specialization
#include<iostream.h>
#include<conio.h>
#include<string.h>
template<class T>
T max(T a, T b)
{
        return a>b?a:b;
```

```
}
char *max(char *a, char *b)
{
        return strcmp(a,b)>0?a:b;
}
int main()
{
        cout<<"max(4,6)="<<max(4,6)<<endl;
        cout<<"max(5.5,3.8)="<<max(5.5,3.8)<<endl;
        cout<<"max(\"Arjun\",\"Gayatri\")="<<max("Arjun","Gayatri")<<endl;
        getch();
        return 0;
}
```

Output

```
max(4,6)=6
max(5.5,3.8)=5.5
max("Arjun","Gayatri")=Arjun
```

11.7 RULES FOR USING TEMPLATES

1. The name of a parameter can appear only once within a formal parameter list. For example:

```
Template <class type>
Type max ( type a, type b)
{
return a > b ? a : b ;
}
```

2. The keyword, class must be specified before the identifier.
3. Each of the formal parameters should form a part of the signature of the function. For example:

```
Template < class T >
Void func ( T a )
{
.....
}
```

4. Templates cannot be used for all overloaded functions. They can be used only for those functions whose function body is the same and argument types are different.
5. A built-in data type is not to be given as a template argument in the list for a template function but the function may be called with it. For example:

```
Template < class ZZ , int > // wrong
Void fn ( ZZ , int) ;
Template < class Z > // right
Void fn ( ZZ , int ) ;
```

6. Template arguments can take default values. The value of these arguments become constant for that particular instantiation of the template.

Self Assessment

Fill in the Blanks.

1. Template functions are blueprints of a function that can be applied to ..... data types.
2. Template can be instantiated for all data types, eliminating duplication of the .....
3. When the function is invoked with actual parameters, the identifier type is ..... with the actual type of the parameter.
4. The process of substitution, depending on the parameter type passed to the function, is called ..... instantiation.
5. We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the .....brackets.
6. C++ Class Templates are used where we have multiple copies of code for different data types with the same .....

Exercise

1. How the use of templates increases code reusability? What happens when functions having same logic with different data types are implemented by using templates?
2. Write a program to find sum of two numbers with different data types by using template functions.
3. In what case it is better to define class templates? Explain the use of class templates with suitable example
4. What happens during template instantiation? Describe the rules of using templates with small code snippets.
5. When template specialization is necessary? Write a class template that adds any type of data members and then define class template specialization that concatenates the data members when the members are of type string.

Chapter 12

Input/Output with Files

Objectives

After studying this unit, you will be able to:

- Demonstrate the opening, reading/writing and closing a file.
- Describe the appending and other operations in a file.
- Recognize Binary and Text files.
- Understand random access operations in files.

12.1 INTRODUCTION

All programs require some input and produce some output but those input and output are lost as the program execution terminates. Files are required to save our data (both input and output) for future use, as RAM is not able to hold our data permanently. Programs would not be very useful if they cannot input and/or output data from/to users. Some programs that require little or no input for their execution are designed to be interactive through user console - keyboard for input and monitor for output. However, when data volume is large it is generally not convenient to enter the data through console. In such cases data can be stored in a file and then the program can read the data from the data file rather than from the console.

The data may be organized in fixed size record or may be in free form text. The various operations possible on a data file using C++ programs are:

1. Opening a data file
2. Reading data stored in the data file
3. Writing/Appending data from a program into a data file
4. Saving the data file onto some secondary storage device
5. Closing the data file once the ensuing operations are over
6. Checking status of file operation

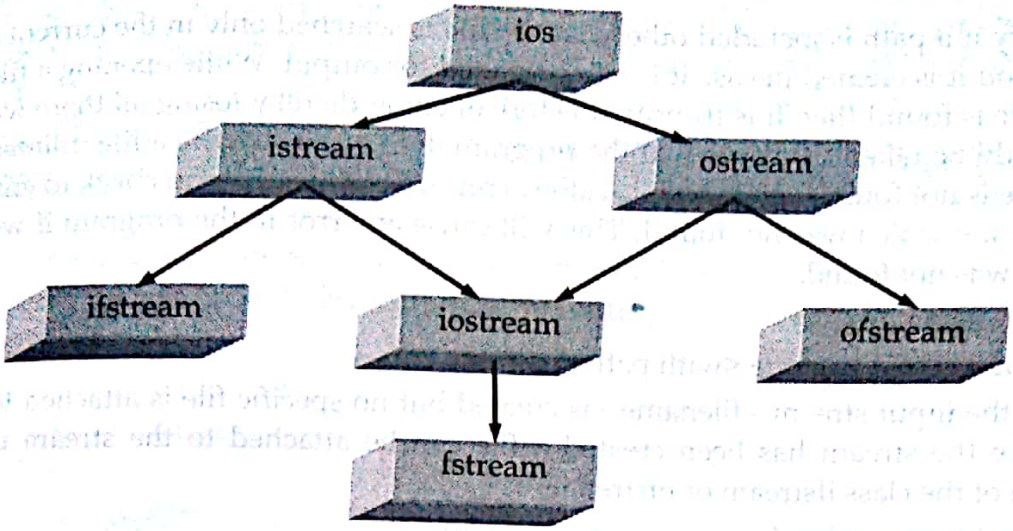
12.2 STREAMS CLASS HIERARCHY

C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and from the program. C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes. Often the array representing a data file is indexed from 0 to len-1, where len is the total number of bytes in the entire file. A file normally remains in a closed state on a secondary storage device until explicitly opened by a program. A file in open state file has two pointers associated with it. These two file positions are independent, and either one can point anywhere at all in the file.

a. **Read pointer:** This pointer al ways points to the next character in the file which will be read if a read command is issued next. After execution of each read command the read pointer moves to point to the next character in the file being read.

b. **Write pointer:** The write pointer indicates the position in the opened file where next character being written will go. After execution of each write command the write pointer moves to the next location in the file being written.

The Language like C/C++ treat every thing as a file, these languages treat keyboard, mouse, printer, Hard disk, Floppy disk and all other hardware as a file. The Basic operation on text/binary files are: Reading/writing, reading and manipulation of data stored on these files. Both types of files need to be open and close. C++ has support both for input and output with files through the following classes:



fstream: The fstream class is an iostream derivative specialized for combined disk file input and output. Its constructors automatically create and attach a filebufbuffer object. Although the **filebuf** object's get and put pointers are theoretically independent, the get area and the put area are not active at the same time.

b. **ofstream:** A stream class for formatted or unformatted file output. The ofstream class can be used for files which are to be used only for output. The ofstream class provide no additional capabilities over the fstream class. Ofstream class used for writing operations is derived from ostream class.

c. **ifstream:** A stream class for formatted or unformatted file input. The ifstream class can be used for files which are to be used only for input. The ifstream class provide no additional capabilities over the fstream class. Ifstream class used for writing operations is derived from istream class.

I/O streams are created and maintained in the RAM. Therefore, when dealing with the output file stream, the data is not saved in the file as the program enters them. A buffer in the memory holds the data until the time you close the file or the buffer is full. When you close the file the data is actually saved in the designated file on the disk. Once the data has been written to the disk the buffer becomes empty again. In case you want to force the data be saved even though the buffer is not full without closing the file you can use the flush() function. A call to flush() function forces the data held n the buffer to be saved in the file on the disk and get the buffer empty.

12.3 OPENING A FILE

The first operation generally done on an object of one of the stream classes is to associate it to a real ile, that is to say, to open a file. The open file is represented within the program by a stream object an instantiation of one of these classes) and any input or output performed on this stream object will e applied to the physical file. A data file can be opened in a program in many ways. These methods re described below:

ifstream filename("filename <with path>");

Or ofstream filename("filename <with path>");

his is the form of opening an input file stream and attaching the file "filename with path" in one ingle step. This statement accomplishes a number of actions in one ste:

1. Creates an input or output file stream
2. Looks for the specified file in the file system
3. Attaches the file to the stream if the specified file is found otherwise returns a NULL value

case the file has been successfully attached to the stream the pointer is placed at the first position. he file stream created is accessible using the object's name. The specified file is searched in the

specified directory if a path is included otherwise the file is searched only in the current directory. If the file is not found it is created in case it is being opened for output. While opening a file for output if the specified file is found then it is truncated before opening thereby losing all there was in the file before. Care should be taken to ensure that the program does not overwrite a file unintentionally. In any case if the file is not found then a NULL value is returned which we can check to ensure that we are not reading a file which was not found. This will cause an error in the program if we attempt to read a file which was not found.

```
ifstream filename;
filename.open("file name <with path>");
```

In this approach the input stream - filename - is created but no specific file is attached to the stream just created. Once the stream has been created a file can be attached to the stream using open() member function of the class ifstream or ofstream.

```
ifstream filename(char *fname, int open_mode);
```

In this form the ifstream constructor takes two parameters - a filename and another the mode in which the input file would be read. C++ offers a host of different opening modes for the input file each offering different types of reading control over the opened file. The file opening modes have been implemented in C++ as enumerated type called ios. The various file opening modes are listed below.

Opening Mode	Description
ios::in	Open file for reading
ios::out	Open file for writing
ios::ate	Initial position: end of file
ios::app	Every output is appended at the end of file
ios::trunk	If the file already existed it is erased
ios::binary	Binary mode

These flags can be combined using bitwise OR(|) operator. 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 function-member

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

All of the member functions open of classes **ofstream**, **ifstream** and **fstream** include a default mode when opening files that varies from one to the other: class defaumode to parameter

File StreamClass	Default Opening Mode
Ofstream	ios::out | ios::in
Ifstream	ios::in
Fstream	ios::out | ios::in

The default value is only applied if the function is called without specifying a *mode* parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined. Since the first task that is performed on an object of classes **ofstream**, **ifstream** and **fstream** is frequently to open a file, these three classes include a constructor that directly calls the **open** member function and has the same parameters as this. This way, we could also have declared the previous object and conducted the same opening operation just by writing:

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

Both forms to open a file are valid. You can check if a file has been correctly opened by calling the member function is_open(). That returns a bool type value indicating true in case that indeed the object has been correctly associated with an open file or false otherwise.

12.4 CLOSING A FILE

When reading, writing or consulting operations on a file are complete we must close it so that it becomes available again. In order to do that we shall call the member function close(), that is in charge of flushing the buffers and closing the file. Its form is quite simple:

```
void close ();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes. In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close.

12.5 UNFORMATTED INPUT/OUTPUT

Unformatted input/output is most basic form of input/output. It transfers internal representation of data directly between memory and file. Following member functions of istream class and ostream class are widely used for performing unformatted input/output.

- → get, put, and getline methods
- → read, write methods

12.5.1 Reading Data by getline() Method

With extraction operator reading terminates after reading white space character therefore above program is able to read single word from file. We can overcome above problem by using getline() function as below:

EXAMPLE

```
//Program to read the content of file using " fin.getline(str,79);"
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
int main()
{
        ifstream fin;       char str[80];
        fin.open("abc.txt");
        fin.getline(str,79);
        /* reads only first line from file as new line is treated as termination point. Reading
        also terminates when 79 characters are read. */
        cout<<"\n From File :"<<str;
        getch();
        return 0;
}
```

Output

```
From File: This is my first program in file handling
```

12.5.2 Using get and put Methods and Detecting End of File

The class ifstream has a member function eof() that returns a nonzero value if the end of the file has been reached. This value indicates that there are no more characters in the file to be read further. This

function is therefore used in the while loop for stopping condition. End of file can be detected in two ways: *Using EOF( ) member function and Using filestream object.*

Detecting End of File Using EOF() member function

Syntax

```
Filestream_object.eof( );
```

EXAMPLE

```
//Detecting end of file with cin.get()
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
int main()
{
        char ch;
        ifstream fin;
        fin.open("abc.txt");
    while(!fin.eof())   // using eof() function
    {
        fin.get(ch); // Reads one character at a time
        cout.put(c); //equivalent cout<<ch;
    }
        fin.close();
        getch();
        return 0;
}
```

Output

```
This is my first program in file handling
Hello again
```

Detecting End of File Using Filestream Object

EXAMPLE

```
//Detecting end of file using filestream object
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
int main()
{
        char ch;
        ifstream fin;
        fin.open("abc.txt");
        while(fin) // file object
        {
                fin.get(ch);
                cout<<ch;//or cout.put(c)
        }
```

```
    fin.close();
    getch();
    return 0;
}
```

Output

```
This is my first program in file handling
Hello again
```

EXAMPLE

```
/*Program to read the contents of a text file and display them on the screen using insertion
operator and getline method.*/
#include<fstream.h>
#include<conio.h>
#include<iostream.h>
int main()
{
    char str[100];
    ifstream fin;
    fin.open("abc.txt");
    while(!fin.eof())
    {
        fin.getline(str,99);
        cout<<str;
    }
    fin.close();
    getch();
    return 0;
}
```

Output

```
This is my first program in file handling Hello again
```

EXAMPLE

```
/*Program to read the contents of a text file and display them on the screen using
extraction operator */
#include<fstream.h>
#include<conio.h>
#include<iostream.h>
int main()
{
    char str[20];
    ifstream fin;
    fin.open("abc.txt");
    while(!fin.eof())
    {
        fin>>str;//Reads one word at a time
```

```
            cout<<str<<endl;
        }
        fin.close();
        getch();
        return 0;
}
```

Output

```
This
is
my
first
program
in
file
handling
Hello
again
```

12.5.3 Reading and Writing by Using Read () and Write Member Functions

File streams include two member functions specifically designed to input and output binary data sequentially: *write and read*. The first one (write) is a member function of ostream class inherited by ofstream and read is a member function of istream class that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

Syntax for write( )

```
Fileobject.write( (char *)&object, sizeof(object));
```

Syntax for read( )

```
Fileobject.read( (char *)&object, sizeof(object));
```

EXAMPLE

```
//Program to write data by using write() member function
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
struct student
{
        int roll ;
        char name[30];
        char address[60];
};
int main()
 {
        student s;
        ofstream fout;
        fout.open("student.dat");
        cout<<"\n Enter Roll Number :";
        cin>>s.roll;
```

```
        cout<<"\n Enter Name :";
        cin>>s.name;
        cout<<"\n Enter address :";
        cin>>s.address;
        fout.write((char *)&s,sizeof(student));
        cout<<"Data is written to the file student.dat"<<endl;
        fout.close();
        cout<<"Done"<<endl;
        getch();
        return 0;
}
```

Output

```
        Enter Roll Number:1
        Enter Name: Sanu
        Enter address: Ktm
        Data is written to the file student.dat
        Done
```

EXAMPLE

```
//Program to read data from a binary File using read ( ) member function
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
struct student
{
    int roll ;
    char name[30];
    char address[60];
};
int main()
{
   student s;
   ifstream fin;
   fin.open("student.dat");
   fin.read((char *)&s,sizeof(student));
   cout<<"\n Roll Number:"<<s.roll;
   cout<<"\n Name:"<<s.name;
   cout<<"\n Address:"<<s.address;
   fin.close();
   getch();
   return 0;
}
```

Output

```
Roll Number:1
Name:Ram
Address:Ktm
```

Reading and Writing Objects

EXAMPLE

```
//Program to write multiple objects to and from file
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class student
{
        int roll ;
        char name[30];
        char address[60];
        public:
        void read_data( );  // member function prototype
        void write_data( ); // member function prototype
};
void student::read_data( )     // member function defintion
 {
        cout<<"\n Enter Roll :";
        cin>>roll;
        cout<<"\n Student name :";
        cin>>name;
        cout<<"\n Enter Address :";
        cin>>address;
 }
void student:: write_data()
 {
        cout<<"\n Roll :"<<roll;
        cout<<"\n Name :"<<name;
        cout<<"\n Address :"<<address;
 }
int main()
 {
        student s;
        int i;
        ofstream fout;
        fout.open("student.dat");
        for(i=0;i<5;i++)
        {
                s.read_data();
                fout.write((char *)&s,sizeof(student));
        }
        fout.close();
        cout<<"Write Completed:"<<endl;
        getch();
        return 0;
}
```

Output

```
Enter Roll :1
Student name :Sandhya
Enter Address :Mnr
Enter Roll :2
Student name :Umesh
Enter Address :Mnr
Enter Roll :3
Student name :Manish
Enter Address :Ktm
Enter Roll :4
Student name :Indra
Enter Address :Pkr
Enter Roll :5
Student name :Bhupendra
Enter Address :Ktm
Write Completed
```

EXAMPLE

```
//Program to read multiple objects from file
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class student
{
        int roll ;
        char name[30];
        char address[60];
        public:
        void read_data( );  // member function prototype
        void write_data( ); // member function prototype
};
void student::read_data( )     // member function defintion
{
        cout<<"\n Enter Roll :";
        cin>>roll;
        cout<<"\n Student name :";
        cin>>name;
        cout<<"\n Enter Address :";
        cin>>address;
}
void student:: write_data()
{
        cout<<"\n Roll :"<<roll;
        cout<<"\n Name :"<<name;
```

```
        cout<<"\n Address :"<<address;
}
int main()
{
        student s;
        int i;
        ifstream fin;
        fin.open("student.dat");
        for(i=0;i<5;i++)
        {
                fin.read((char *)&s,sizeof(student));
                s.write_data( );
        }
        fin.close();
        getch();
        return 0;
}
```

Output

```
Roll :1
Name :Sandhya
Address :Mnr
Roll :2
Name :Umesh
Address :Mnr
Roll :3
Name :Manish
Address :Ktm
Roll :4
Name :Indra
Address :Pkr
Roll :5
Name :Bhupendra
Address :Ktm
```

12.6 FORMATTED INPUT/OUTPUT

Formatted output converts internal binary representation of data into ASCII format and writes data to the file and formatted input reads data from file and converts it into internal binary format. Formatting of data can be done in two ways:

- → By using Manipulators
- → By using overloaded Stream Operators

12.6.1 Formatted Input/Output by Using Overloaded Stream Operators

The ostream's << stream insertion operator is overloaded to convert a numeric value from its internal representation to the text form. By default, the values are displayed with a field-width just enough to hold the text, without additional leading or trailing spaces. You need to provide spaces between the values, if desired.

- For integers, all digits will be displayed, by default.

- For floating-point numbers, the default precision is 6 digits, except that the trailing zeros will not be shown.
- Boolean values are displayed as 0 or 1 by default, instead of true or false.

EXAMPLE

```
//Program to write content to the file
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
int main( )
{
        ofstream fout;
        fout.open("abc.txt");
        fout<<"This is my first program in file handling";
        fout<<"\n Hello again";
        fout.close();
        cout<<"Done"<<endl;
        getch();
        return 0;
}
```

Content of abc.txt file

```
This is my first program in file handling
Hello again
```

EXAMPLE

```
//Program to read content from file
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
int main()
{
        ifstream fin;
        char str[80];
        fin.open("abc.txt");
        fin>>str;
        /* reads only first string from file as spaces is treated as termination point*/
        cout<<"\n From File :"<<str;
        getch();
        return 0;
}
```

Output

```
From File:This
```

12.6.2 Formatted Input/Output by Using Manipulators

Manipulators are operators used to format the output to be displayed in monitor or files. These operators are used in conjunction with insertion (<<) and extraction (>>) operators. Manipulators are defined in iomanip.h header file. We have already discussed about manipulators in chapter 2. We can also use these manipulators with file streams. Some of the widely used manipulators are:

- → setprecision: used to set the precision of floating point numbers
- → setw: used to set the field width
- → setfill: used to set the fill character
- → endl: used to insert new line character

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#include<string.h>
int main()
{
        int i,l;
        int m[5];
        char sub[5][20]={"Mathmatics","English","C Prog",
                                "Statistics","Sociology"};
        float total=0.0,per;
        ofstream fout;
        fout.open("result.txt");
        cout<<"Enter marks of Mathematics, English, C Prog, Statistics
        and Sociology"<<endl;
        for(i=0;i<5;i++)
        {
            cin>>m[i];
        }
        fout<<"\t"<<"Subject"<<setw(15)<<"Marks"<<endl;
        fout<<"\t"<<"--------------------"<<endl;
        for(i=0;i<5;i++)
        {
                total=total+m[i];
                l=strlen(sub[i]);
                fout<<"\t"<<sub[i]<<setw(22-l)<<m[i]<<endl;
        }
        per=total/5;
        fout<<"\t"<<"--------------------"<<endl;
        fout<<"\t"<<"Total"<<setw(17)<<total<<endl<<"\t";
        fout<<"Percentage"<<setw(12)<<setprecision(2)<<per<<endl;
        cout<<"Write Completed"<<endl;
        return 0;
}
```

Content of result.txt

```
Subject                 Marks
---------------------
Mathematics
English                  67
C Prog                   58
Statistics               89
Sociology                67
                         56
```

Total	337
Percentage	67.40

12.7 RANDOM FILE ACCESS

Till now our discussion is about sequential file access. Each type of iostream has a concept of where its "next" character will come from (if it's an istream) or go (if it's an ostream). In some situations, you might want to move this stream position. You can do so using two models: one uses an absolute location in the stream called the streampos; the second works like the standard C library functions fseek( ) for a file and moves a given number of bytes from the beginning, end, or current position in the file.

The streampos approach requires that you first call a "tell" function: tellp( ) for an ostream or tellg( ) for an istream. (The "p" refers to the "put pointer," and the "g" refers to the "get pointer.") This function returns a streampos which you can later use in calls to seekp( ) for an ostream or seekg( ) for an istream. The second approach is a relative seek and uses overloaded versions of seekp( ) and seekg( ). The first argument is the number of characters to move: it can be positive or negative. The second argument is the seek direction. Some important member functions are:

→**seekg( ):** It is used to move reading pointer forward and backward

Syntax

```
fileobject.seekg( no_of_bytes,mode);
```

Examples:

```
fin.seekg(50,ios::cur);/*move 50 bytes forward from current
                                                position*/
fin.seekg(50,ios::beg); /* move 50 bytes forward from
                                                beginning*/
fin.seekg(50,ios::end);  /* move 50 bytes forward from end .
```

→seekp( ): It is used to move writing pointer forward and backward

Syntax

```
fileobject.seekp(no_of_bytes, mode);
```

Examples:

```
fout.seekp(50,ios::cur); /* move 50 bytes forward from current
                                                position*/

fout.seekp(50,ios::beg);  /*move 50 bytes forward from
                                                beginning*/

fout.seekp(50,ios::end); /* move 50 bytes forward from end
```

→**tellp( ):** It return the distance of writing pointer from the beginning in bytes

Syntax

```
Fileobject.tellp( );
```

Example:

```
long  n = fin.tellp( );
```

→ **tellg( ):** It return the distance of reading pointer from the beginning in bytes

Syntax

```
Fileobject.tellg( );
```

Example:

```
long  n = fout.tellg( );
```

EXAMPLE

```
//Program to read third object from file student.dat
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class student
{
        int roll ;
        char name[30];
        char address[60];
        public:
        void read_data( );  // member function prototype
        void write_data( ); // member function prototype
};
void student::read_data( )     // member function defintion
{
        cout<<"\n Enter Roll :";
        cin>>roll;
        cout<<"\n Student name :";
        cin>>name;
        cout<<"\n Enter Address :";
        cin>>address;
}
void student:: write_data()
{
        cout<<"\n Roll :"<<roll;
        cout<<"\n Name :"<<name;
        cout<<"\n Address :"<<address;
 }
int main()
{
        student s;
        int i;
        ifstream fin;
        fin.open("student.dat");
        fin.seekg(sizeof(s)*2,ios::cur);
        fin.read((char *)&s,sizeof(student));
        s.write_data( );
        fin.close();
        getch();
        return 0;
}
```

Output

```
Roll :3
```

Name :Manish
Address :Ktm

12.8 TESTING ERRORS DURING FILE OPERATIONS

Sometimes during file operations, errors may also creep in. For example, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such as invalid operation may be performed. There might not be enough space in the disk for storing data. There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

→eof(): Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).

→fail(): Returns non-zero (true) when an input or output operation has failed.

→gad(): Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.

→bad():Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.

12.9 BUFFERS AND SYNCHRONIZATION

When we operate with file streams, these are associated to an internal buffer of type streambuf. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer. When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called synchronization and takes place under any of the following circumstances:

- **When the file is closed:** before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- **Explicitly, with member function sync():** Calling stream's member function sync(), which takes no parameters, causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

Self Assessment

Fill in the Blanks.

1. All programs require some input and produce some output but those input and output are lost as the ..... terminates

2. When data volume is large it is generally not convenient to enter the data through ..... In such cases data can be stored in a file and then the program can read the data from the data file rather than from the .....
3. A stream is a data ..... for input/output of data to and from the program.
4. A file normally remains in a closed state on a secondary storage device until explicitly opened by a program. A file in open state file has ..... pointers associated with it.
5. The fstream class is an iostream derivative specialized for combined disk file input and output. Its constructors automatically create and attach a ..... object.
6. The ifstream class provide no additional capabilities over the fstream classś. Ifstream class used for writing operations is derived from ..... class.
7. The open file is represented within the program by a stream object (an instantiation of one of these classes) andcf 4rcfffw3aa any input or output performed on this stream object will be applied to the ..... file.
8. The specified file is searched in the specified directory if a path is included otherwise the file is searched only in the ..... directory.
9. When reading, writing or consulting operations on a file are complete we must ..... it so that it becomes available again.
10. Binary mode file contains information in the same format as it is held in memory. In binary files, no delimiters are used for a line and no ..... occur here.

Exercise

1. What is stream? Explain the different stream classes used in c++ for input and output with their use.
2. What are the different operations that can be performed on files? Describe each of the operation with proper syntax.
3. Write a program to read eid, name and salary of 10 students from keyboard and write them into a file 'emp.doc'.
4. How binary files are different from character files? Compare and contrast between them with their strength and drawback.
5. Write a program to write records of 5 items in the file 'item.dat'. Item class should contain data members code and price.
6. What is meant by random access to files? Describe the different functions that supports random access to files
7. Write a program to read first, third and fifth record from the file 'item.dat' and display them in monitor.
8. Write a program to calculate number of Objects in a file.

❑❑❑

Model Question

Tribhuvan University

Institute of Science and Technology

Course Title: Object Oriented Programming
Course No: CSC161
Level: B. Sc CSIT First Year/ Second Semester

Full Marks: 60
Pass Marks: 24
Time: 3 Hrs

Section A

Long Answer Questions

Attempt any two questions. **[2x10=20]**

1. What is object oriented programming? Explain objects, class, encapsulation, data hiding, inheritance, and polymorphism.
2. Explain operator overloading. Write a program that overloads insertion and extraction operators.
3. What is inheritance? Explain the ambiguities associated with multiple inheritance with suitable example programs.

Section B

Short Answer Questions

Attempt any eight questions. **[8x5=40]**

4. Explain the purpose of a namespace with suitable example.
5. What is the principle reason for passing arguments by reference? Explain with suitable code.
6. Why constructor is needed? Explain different types of constructors with example.
7. Write a program that illustrates the conversions between objects of different classes having conversion function in source object.
8. Explain the difference between private and public inheritance with suitable diagram.
9. Why friend function is required? Discuss with example.
10. How late binding is different from early binding. Write a program that explains late binding using virtual function.
11. Why do we need exceptions? Explain "exceptions with arguments" with suitable program.
12. What are the advantages of using the stream classes for I/O? Write a program that writes object to a file.

•••