SYLLABUS

OBJECT ORIENTED PROGRAMMING AND C++

SECTION A

OOP paradigm, Advantages of OOP, Comparison between functional programming and OOP approach, characteristics of Object oriented Language objects, Class, Inheritance, Polymorphism, and abstraction, encapsulation, Dynamic Binding, Message passing.
Introduction to C++, Indetifier and keywords, constants, C++ Operators, Type conversion, variable declaration, Statement, expressions, User defined data types, Conditional expression (For, While, Do-while) loop statements, breaking control statements (Break, Continue).

SECTION B

Defining a function, types of functions, Inline functions, Call by value and Call by reference, Preprocessor, Header files and standard functions, Structures, Pointers and structures, Unions, Enumeration.

SECTION C

Classes, Member functions, Objects, Array of objects, Nested classes, Constructors, Copy constuctors, Destructors, Inline member functions, Static class member, friend functions, Dynamic memory allocation.
Inheritance: Single inheritance, Multi-level inheritance, Hierarchical, Virtual base class, Abstract classes, Constructors in Derived classes, Nesting of classes.

SECTION D

Function overloading, Operator overloading, Polymorphism, Early binding, Polymorphism with pointers, Virtual functions, Late binding, Pure virtual functions, Opening and closing of files, Stream member functions, Binary file operations, Structures and file operations, classes and file operations, Random access file processing.

SECTION A

UNIT 1 OOP PARADIGM AND INTRODUCTION TO C++

NOTES

★ LEARNING OBJECTIVES ★

- Object-Oriented Programming Paradigm
- Benefits of OOP
- Characteristics of Object-Oriented Language
- Introduction to C++
- Identifier, Keyboards and Constants
- Variable Declaration
- C++ Operators
- Statement and Expressions
- User Defined Data Types
- Conditional Expression
- Loop Statements
- Breaking and Control Statements

OBJECT-ORIENTED PROGRAMMING PARADIGM

Introduction

Software products should always be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are:

- Correctness
- Maintainability
- Reusability
- Openness and interoperability
- Portability
- Security
- Integrity
- User friendliness

To build today's complex software, it just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate ***sound construction techniques and program structures that are easy to comprehend, implement and modify.***

Object-Oriented Programming Paradigm

Object oriented programming treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to functions that operate on it and protects it from accidental modifications from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The *data of an object can be accessed only by the functions associated* with that object. However, a function of one object can access the function of other objects.

Some of the striking features of OOP are:

- Emphasis is on data rather than procedure
- Programs are divided into what we know as objects
- *Data structures are designed such that they characterize the* objects
- Functions that operate on the data of an object are tied together in the data structure
- Data is hidden and cannot be accessed by external functions
- Objects may communicate with each other through the functions
- Follows bottom-up approach in program design.

BENEFITS OF OOP

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing code from scratch.
3. The principle of data hiding helps the programmer build secure programs that cannot be invaded by code in other parts of the program.
4. It is possible to map objects in the problem domain to those in the program.
5. It is easy to partition work in a project-based on objects.

NOTES

Comparison between Functional Programming and OOP Approach

Procedure-Oriented Programming

Conventional programming using high-level languages such as COBOL, FORTRAN and C is commonly known as procedure-oriented programming. Here, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of applications are written to accomplish these tasks. The primary task is a function. The technique of hierarchical decomposition is used to specify the tasks to be completed for solving a problem.

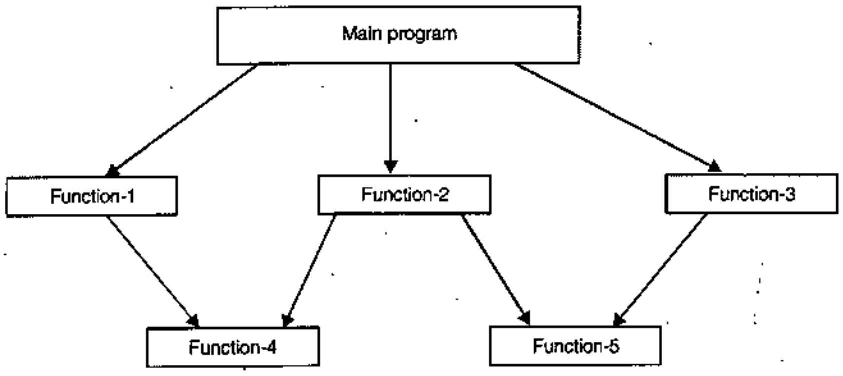


Fig. 1

Procedure-oriented programming consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as ***functions***. We normally use a flowchart to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data.

In a multifunction program, many important data items are placed as *global* so that they may be accessed by all the functions. Each function may have its own local data.

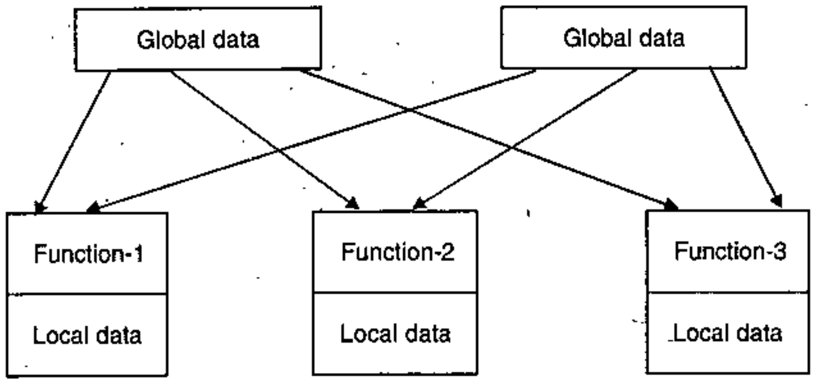


Fig. 2

Some characteristics exhibited by procedure-oriented programming include:

- Emphasis is on doing things (algorithms)

- A **collection object** is an object of a collection, *e.g.,* array, list, set, and bag. A collection holds members. Sample collection objects are listOfStudents and setOfCourses. For example, listOfStudents = {student1, student2}.

Classes

A **class** is a description of a group of objects with similar attributes, common operations, common relationships (association, aggregation, interaction, and generalization specialization) and a common semantic purpose. In S/W a class is a module. An **attribute** is a characteristic or property of an object. An attribute typically holds an atomic object, *e.g.,* an integer, a float, a character, etc. For example, an attribute of a car is gasQuantity which is a float. However, an attribute can hold a structured object or collection object to implement a relationship. For example, an attribute of a car could be currentPassengers which holds a set of current passenger objects. An attribute can hold a set of literals, *e.g.,* a string of characters. An **operation** is a function, action or set of actions. For example, an operation of a car is to "set gas quantity" and "start". A **relationship** is a connection or link between classes or between objects. The primary relationships are association ("has a"), aggregation ("part of"), generalization specialization ("is a"), and interaction (*"calls or communicates"*). *For example, the* Car Class has an association relationship with the Passenger Class. The Car Class has a generalization specialization relationship with the Vehicle Class. The Car Class has an aggregation relationship with the Motor Class. An interaction relationship (messages) exists *between objects of the Car Class and objects of the Motor Class.* The semantic purpose of a class is the reason for being or existence of objects of the class. For example, the semantic purpose of objects of the Car Class is to provide transportation to carry users from one location to another. We model a class with a class diagram and a class specification. Objects are variables of the type *class.*

Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called *class*) is known as ***encapsulation.*** The data is not accessible to the *outside world and only those functions that are wrapped in the class* can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called ***data hiding*** or ***information hiding.***

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes.

They encapsulate all the essential properties of the objects to be created. The attributes are sometimes called ***data members***. The functions that operate on these data are sometimes called ***methods***.

Since classes use the concept of data abstraction, they are known as ***Abstract Data Types (ADT)***

Inheritance

The process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features on existing class without modifying it.

Polymorphism

An operation may exhibit different behaviors in different instances. The behaviour depends upon the types of data used in the operation.

The process of making an operation to exhibit different behaviour in different instances is known as ***operator overloading***. Using a single function name to perform different types of tasks is known as ***function overloading.***

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is used extensively to implement inheritance.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run time. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

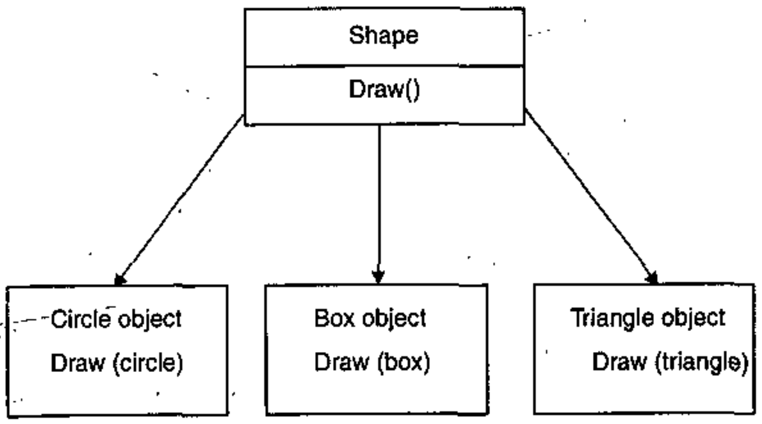


Fig. 4

NOTES

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour
2. Creating objects from class definitions
3. Establishing communication among objects

Objects communicate with each other by sending and receiving information much the same way as people pass messages to one another.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves the specification of the name of the object, the name of the function (message) and the information to be sent.

Object-Oriented Languages

The language should support several of the OOP concepts to claim it is object-oriented. Depending upon the features it supports, they can be classified into the following two categories:

- Object-based programming languages, and
- Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. For, example, Ada.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely inheritance and dynamic binding.

Examples include C++, Smalltalk and Java.

INTRODUCTION TO C++

In early days of computer programming, programmers worked with the most primitive computer instructions say machine language. These instructions were represented by long strings of ones and zeroes.

Soon, assemblers were invented to map machine instructions to human-readable and manageable mnemonics, such as ADD and MOV. Mean time, higher-level languages evolved, such as BASIC, COBOL, C. These languages allow people to work with something approximating words and sentences, such as I = S + 100. These instructions were translated back into machine language by interpreters and compilers. Most of the procedural languages like C was not able to *solve real world problems* using an entity as object and software development process was costly too. As object-oriented analysis, design, and programming began to attract the software industry, Bjarne Stroustrup took the most popular language for commercial software development, C++, and extended it to provide the features needed to facilitate object-oriented programming. He created C++, and in less than a decade it has gone from being used by only a handful of developers at AT&T to being the programming language of choice for an estimated one million developers worldwide soon. Now C++ is a predominant language for commercial software development. While it is true that C++ is a superset of C, and that virtually any legal C program is a legal C++ program, the leap from C to C++ is very significant. C++ benefited from its relationship to C for many years, as C programmers can easily start programming in C++.

C++ is a extension of C, but it does not mean that you should learn C first. It is unnecessary to learn C first one can easily start C++ programming because C++ allow you to write code in C style. You can learn C++ without prior experience of C. Even if you have no programming experience of any kind you can be a good C++ programmer.

IDENTIFIER, KEYWORDS AND CONSTANTS

A programmer used C++ tokens to write a C++ program. You can compare it with writing a English sentence there you need noun, verbs, punctuations, symbols to write a proper sentence same is true with a C++ program. You will use C++ tokens to write a program. C++ uses the following types of tokens:

- Identifiers
- Keywords
- Constants
- Operators

Identifiers in C++

An identifier is a user defined name , given to various data items in a program like empname, sturollnumber, bookno etc., If you want to use an identifier in a program use following rules:

NOTES

```
void main()
{
int num1
int num2;
cout<<enter values of num1 and num2 ";
cin>>num1;
cin>>num2;
cout<<"first number is:"<<num1;
cout<<"second number is:"<<num2;
cout<<"sum of numbers is :"<< num1+num2;
}
```

Float Data Type

The floating-point number is another data type in the C++ language. Unlike an integer number, a floating-point number contains a decimal point. For instance, 7.01 is a floating-point number; so are 5.71 and –3.14. A floating-point number is also called a real number. A floating-point number is specified by the float keyword in the C++ language.

Like an integer number, a floating-point number has a limited range. The ANSI standard requires that the range be at least plus or minus 1.0*10e37. Normally, a floating-point number is represented by taking 32 bits. Therefore, a floating-point number in C++ is of at least six digits of precision. That is, for a floating-point number, there are at least six digits (or decimal places) on the right side of the decimal point.

Declaring Floating-Point Variables

The following shows the declaration format for a floating-point variable:

```
float variablename;
```

Similar to the character or integer declaration, if you have more than one variable to declare, you can either use the format like this:

```
float variablename1;
float variablename2;
float variablename3;
```

or like the following one:

```
float variablename1, variablename2, variablename3;
```

Program

```
Printing out floating(decimals)numbers on the screen.
/* Printing out float numbers */
#include <iostraem.h>
```

```
void main()
{
float num1
float num2;
cout<<"enter values of num1 and num2 with decimals
   like 2.3";
cin>>num1;
cin>>num2;
cout<<"first  number  is:"<<num1;
cout<<"second  number  is:"<<num2;
cout<<"sum of numbers is :"<< num1+num2;
}
```

Double Data Type

In the C++ language, a floating-point number can also be represented by another data type, called the double data type. In other words, you can specify a variable by the double keyword, and assign the variable a floating-point number.

The difference between a double data type and a float data type is that the former normally uses twice as many bits as the latter. Therefore, a double floating-point number is of at least 10 digits of precision, although the ANSI standard does not specify it for the double data type.

C++ OPERATORS

Operators In C++

An operator is a symbol that instructs C++ to perform some operation, or action, on one or more operands. An operand is something that an operator acts on. In C++, all operands are expressions.

Like c = a + b;

Here + is an operator, while a and b are operands.

C++ offers following operators:

1. The assignment operator
2. Mathematical operators
3. Relational operators
4. Logical operators
5. Address of operator (&)

6. Scope Resolution operator (: :)
7. Bitwise operators

Assignment Operator

NOTES

The assignment operator is the equal sign (=). Its use in programming is somewhat different from its use in regular math. If you write

```
x = y;
a = 560;
```

in a C++ program, it doesn't mean "x is equal to y." Instead, it means "assign the value of y to x." In a C++ assignment statement, the right side can be any expression, and the left side must be a variable name. Thus, the form is as follows:

```
variable = expression;
```

When executed, expression is evaluated, and the resulting value is assigned to variable.

Mathematical Operators

A C++ mathematical operators perform mathematical operations such as addition and subtraction. C++ has two unary mathematical operators (++, - -) and five binary mathematical operators (+ , - , * , / , %).

Program

```
/* Demo of unary operators */
#include <iostream.h>
int a, b;
void main()
{
   /* Set a and b both equal to 5 */
   a = b = 5;
   /* Print them, decrementing each time. */
   /* Use prefix mode for b, postfix mode for a */
   cout<< a- - << - -b;
   cout<<a - - < - -b;
   cout<<- - a<<- -b;
   }
```

Use of Binary Operators in C++ program

Program

```
/* Use of  Binary Operators in C++ program */
#include <iostream.h>
```

```
void  main()
{
int x;
int y;
int a,b,c,d,e;
cout<<"enter values of x and y ";
cin>>x>>y ;
a=x+y;
b=x-y;
c=x*y;
d=x/y;
e=x%y;
cout<< a;
cout<< b;
cout<< c;
cout<< d;
cout<< e;
}
```

if x=50 and y=3 then

```
a=53
b=47
c=150
d=16
e=2
```

Relational Operators

C++ relational operators are used to compare expressions, asking questions such as, "Is a greater than 10?" or "Is y equal to 0?" An expression **RELATIONAL OPERATOR** alse (0). C++ has six relational operators:

Operator	*Symbol*	*Example*
Equal	==	x==y
Greater than	>	x > y
Less than	<	x < y
Greater than or equal to	>=	x >= y
Less than or equal to	<=	x <= y
Not equal	!=	x!= y

Program

```
/* Demo of  relational expressions */
   #include <iostream.h>
   int x;
   void main()
   {
      x=(7 == 7);          /* Evaluates to 1 */
      cout<<"\n a=(7==7)\n a="<<x;
      x=(7!=7);                /* Evaluates to 0 */
      cout<<" \n a=(7!=7)\n a="<< x;
      x=(12==12)+(7!=1);  /* Evaluates to 1+1 */
      cout<<"\na=(12==12)+(7!=1)\na=\n"<<x;
   }
```

Logical Operators

Sometimes you might need to use more than one relational question at once. For example, "If x is Male, have age more than 40 and not a graduate", C++ logical operators allow you to combine two or more relational expressions into a single expression that evaluates to either true or false.

Operator	*Symbol*	*Example*
AND	&&	a=5 && b=7
OR	||	x=56 || y=80
NOT	!	!c='s'

Conditional Operators

The conditional operator is C++ only ternary operator, meaning that it takes three operands. Its syntax is:

```
exp1 ? exp2 : exp3;
```

If exp1 evaluates to true (that is, non-zero), the entire expression evaluates to the value of exp2. If exp1 evaluates to false (that is, zero), the entire expression evaluates as the value of exp3. For example, the following statement assigns the value 1 to x if y is true and assigns 100 to x if y is false:

```
x = y ? 1 : 100;
```

Likewise, to make z equal to the larger of x and y, you could write

```
z = (x > y) ? x : y;
```

Perhaps you've noticed that the conditional operator functions somewhat like an if statement. The preceding statement could also be written like this:

```
    if (x > y)
    z = x;
    else
    z = y;
```

The conditional operator can't be used in all situations in place of an if...else construction, but the conditional operator is more concise. The conditional operator can also be used in places you can't use an if statement, such as inside a single cout.

Statement.

```
    cout<<"The larger value is="<<((x > y) ? x : y) );
```

STATEMENT AND EXPRESSIONS

C++ Statement

A statement is a complete direction instructing the computer to carry out some task. In C++, statements are usually written one per line, although some statements can use multiple lines. C++ statements always end with a semicolon (except for preprocessor directives such as #define and #include). Some of C++ statement for examples are:

```
    x = 2 + 3;
    sum=num1+num2+num3;
```

is an assignment statement. It instructs the computer to add 2 and 3 and to assign the result to the variable x.

White Space

The term white space refers to spaces, tabs, and blank lines in your source code. The C++ compiler isn't sensitive to white space. When the compiler reads a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space. Thus, the statement

```
    x=2+3;
```

is equivalent to this statement:

```
    x = 2 + 3;
```

Null and Compound Statements

Null Statements

If you place a semicolon by itself on a line, you create a null statement—a statement that doesn't perform any action. This is perfectly legal in C++.

Compound Statements

A compound statement, also called a block, is a group of two or more C++ statements enclosed in braces. Here's an example of a block:

```
if (x>3)
{
   cout<<"Hello\n";
   cout<<"world!";
}
```

In C++, a block can be used anywhere a single statement can be used. It's a good idea to place braces on their own lines, making the beginning and end of blocks clearly visible. Placing braces on heir own lines also makes it easier to see whether you've left one out.

USER DEFINED DATA TYPES

In C++, you can use user defined data types like:

- Arrays
- Structures
- Pointers
- Union
- Enumeration

CONDITIONAL EXPRESSION

If A student will score 50% marks in each subject he or she will be declared pass in University exam or If your age is 18 years or more you can use your vote to elect your MP or PM. This type of real life situation need conditional programming or decision making. The if statement enables you to test for a condition (such as whether your percentage of marks >50 or not) and branch to different parts of your code, to process other parts.

The simplest form of an if statement is this:

```
if (expression)
    statement;
```

```
Examples
     A) if(marks>=50)              B) if(Age>=18)
     {                             {
     char result= 'P' ;            char vote= 'y';
     }                             }
     else                          else
```

```
{                                  {
char result= 'F';                  char vote= 'n';
}                                  }
```

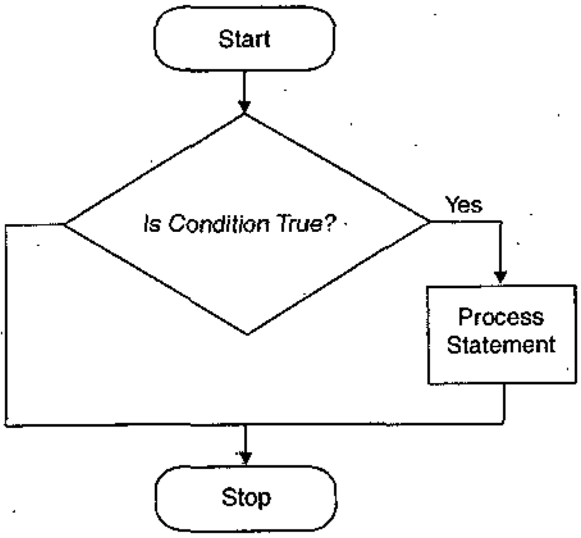


Fig. 5 *Flow chart of If*

You can use multiple statements, as in following example:

```
if(expression)
{
   statement1;
   statement2;
   statement3;
}
if(basic>=8000)
{
da=(basic*67) / 100;
hra=(basic*25)  /100
net  =basic+da+hra;
}
```

If .. Else

You can also ask the compiler to check a condition; if that condition is true, the compiler will execute the intended statement. Otherwise, the compiler would execute alternate statement. This is performed using the syntax:

```
if(Condition)
   Statement1;
else
   Statement2;
```

NOTES

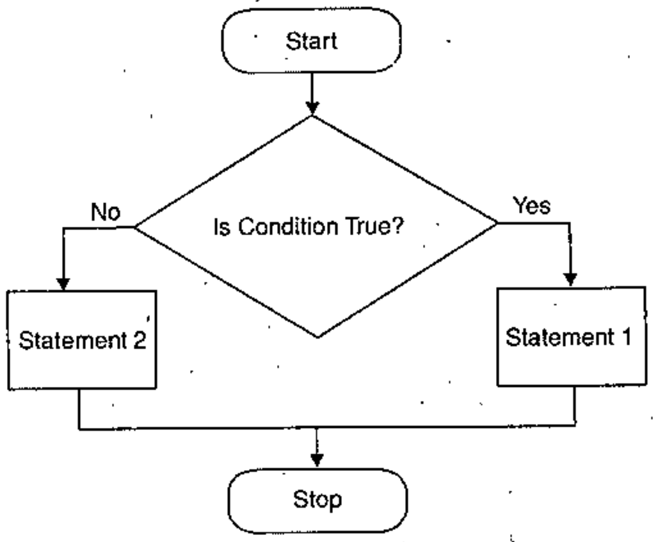


Fig. 6 *Flow chart of nested if*

C++ program to compute netsalary of an employee, netsalary is a sum of basicsalry, da and hra of employee. An employee getting basic salary more than or equal to 8000 will get 67% da, 25% hra on his basic salary otherwise he will get 50% da, 18% hra on his basic salary.

Program

```
/* Use of  if -else  in C++ program */
#include <iostream.h>
void  main()
{
int basic ;
float da, hra, netsalary;
cout<<"enter basic salary of employee ";
cin>>basic;
if(basic>=8000)
{
da= basic*67/100;
hra=basic*25/100;
netsalary=basic+da+hra;
}
else
{
   da= basic*50/100;
   hra=basic*18/100;
netsalary=basic+da+hra;
}
```

```
    cout<<"Netsalary is"<<netsalary;
    }
```

Sometimes, your program will need to check many more than that. The syntax for such a situation is:

```
    if(Condition1)
       Statement1;
    else if(Condition2)
       Statement2;
```

An alternative syntax would add the last else as follows:

```
    if(Condition1)
       Statement1;
    else if(Condition2)
       Statement2;
    else if(Condition3)
       Statement3;
    else
       Statement-n;
```

The compiler will check the first condition. If Condition1 is true, it will execute Statement1. If Condition1 is false, then the compiler will check the second condition. If Condition2 is true, it will execute Statement2. When the compiler finds a Condition-*n* to be true, it will execute its corresponding statement. It that Condition-*n* is false, the compiler will check the subsequent condition. This means you can include as many conditions as you see fit using the else if statement. If after examining all the known possible conditions you still think that there might be an unexpected condition, you can use the optional single else.

Program

```
    //program to check a character input by keyboard
for vowel (a,e,i,o,u)
    /* Use of if -else-if in C++ program */
    #include <iostream.h>
    void main()
    {
    char alpha;
    cout<< " enter a character ";
    cin>>alpha;
    if (alpha=='a')
    {
```

NOTES

```
cout<< "A Vowel ";
}
   else if (alpha = = 'e')
   {
   cout<< "A Vowel ";
   }
   else if (alpha = = 'i')
   {
     cout<< "A Vowel ";
       }
       else if (alpha = = 'o')
       {
         cout<< "A Vowel ";
           }
           else if (alpha == 'u')
              {
              cout<< "A Vowel ";
                 }
                      else
              cout<< "Not a Vowel ";
}
```

Program

```
#include <iostream.h>
void main()
{
   char Answer;
   cout << "Are you more than 18 (y=Yes/n=No)? ";
   cin >> Answer;
   if( Answer == 'y' )
   {
      cout << "\n You are mature now ";
      cout << "\nYou can vote in election \n";
   }
   else // Any other answer
   cout << "\nWait for more years \n";
   }
```

Multiple Condition with if

In many programming problems like result of a students, there can be multiple conditions to allot grade based on percentage of marks secured by the student. In C++ to join multiple conditions you can use && (and) operator or you can use || (or) operator. && is used when both conditions are true while || is used when anyone of condition is true. && and || also known as logical operators.

Example.

```
if ( temp<=35  && temp >=15)            // both must   be true
if(alpha =  ='a' || alpha = = 'e' || alpha = = 'i || alpha = = 'o' || alpha
= = 'u')
```

Program

```
      /* Use of || operator with if -else in C++ program */
      #include  <iostream.h>
      void main()
      {
      char alpha;
      cout<< " enter a character  ";
      cin>>alpha;
      if(alpha =='a'||alpha =='e'||alpha =='i|| alpha
=='o'      || alpha =='u')
      {
         cout<< "A Vowel ";
      }
      else
      {
            cout<< "Not a Vowel ";
      }
      }
```

Program

```
      /* Use of  || operator with if -else   in C++ program */
      #include  <iostream;h>
      void  main()
      {
      int percentage ;
      cout<< "Enter percentage of student (1-100)  ";
      cin>>percentage;
      if (per <=40)
```

NOTES

```
{
char grade='F';
}
else if ( per>40 && per <=50)
{
char grade='D';
}
else if ( per>50 && per <=60)
{
char grade='C';
}
else if ( per>60 && per <=70)
{
char grade='B';
}
else
{
char grade='A';
}
cout<< "Grade of student is "<<grade ; }
```

Conditional Operator (?:)

In C ++, the conditional operator (?:) can be used in place of if-else statement to check conditions. This is only one operator in C++ which uses three operands. Syntax is

Condition? expression1 : expression2

x > y ? 10 : 50

Example

```
/* Use of  ?:   in C++ program */
#include <iostream.h>
void main()
{
int x, y  ;
cout<< "Enter two numbers ";
cin>>x>>y;
int  z = x > y?  x : y ;
cout<< "The bigger number is "<< z;
}
```

LOOP STATEMENTS

Sometimes you want to perform an action again and again, like you want to print first 100 even numbers, then loop concept of C++ will help to do it in less numbers of line of codes. In examination processing system same logic has to repeated for *n* numbers of students. In a clock seconds, minutes and hours needles make a loop of 60 cycles. Looping, also called iteration, is used in programming to perform the same set of statements over and over until certain specified conditions are met.

Three statements in C++ are designed for looping:

1. The for statement
2. The while statement
3. The do-while statement

For Loop

The **for** statement is typically used to count a number of items. At its regular structure, it is divided in three parts. The first section specifies the starting point for the count. The second section sets the counting limit. The last section determines the counting frequency. The syntax of the for statement is:

```
for (expression1; expression2; expression3)
{
    statement1;
    }
```

Example

```
    /* Squre & Cube  from1 to 15  numbers */
    #include <iostream.h>
    void main()
    {
    cout<<"Number"<< "Squre "<<" Cube\n ";
    for (int i=1; i<=15; i++)
    {
    cout<<i<<i*i<<i*i*i ;
cout<<"\n";
    }
    }
```

Multiple Values in a for Loop

Example

```
/* Multiple expressions in for loop */
```

```
#include <iostream.h>
void main()
{
   int i, j;
   for (i=0, j=8; i<8; i++, j--)
   cout<<i<<j<<i+j;
   }
```

NOTES

Nested for Loop

A for statement can be executed within another for statement. This is called nesting. By nesting for statements, you can solve complex programming problems.

Example

```
/* Dem of  nesting two for statements */
#include <iostream.h>
void mybox( int, int);
void main()
{
   mybox( 5, 25 );
}
void mybox( int row, int column )
{
   int col;
   for ( ; row > 0; row—)
{
      for (col = column; col > 0; col—)
{
      cout<<"*";
}
   cout<<"\n";
}
}
```

While Loop

The while statement, also known as while loop, executes a block of statements as long as a specified condition is true. The while statement has the following syntax:

```
while (condition)       while(x<=10)
{                       {
statement;              cout<<x;
statement;              x=x+2;
increment;              }
}
```

NOTES

Example

```
/* Demo of  simple while statement */
#include <iostream.h>
int count;
void main()
{
```

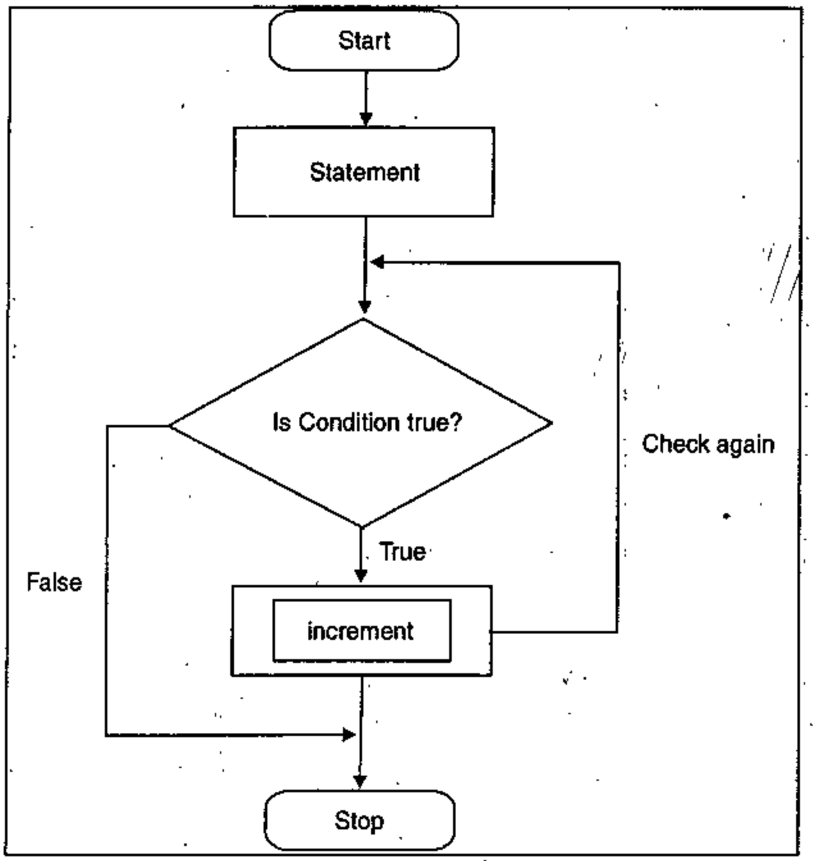


Fig. 7 *While Loop*

```
/* Print the numbers 1 through 25 */
   count = 1;
   while (count <= 25)
   {
      cout<<"Number="<<count;
```

```
             count++;
          }
          }
```

Example

```
#include  <iostream.h>
int number;
void  main()
{
int x = 1;
cout<<"enter the number";
cin>>number;
while (x < 10)
{
number=number * x ;
cout<<number<<"\n");
   x++;
}
```

Do-While Loop

One more loop in C++ is *do...while loop, which executes a block of* statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop.

The *syntax of do...while loop* is:

```
do
{
statement;
increment / decrement in loop ;
} while (condition);
int x=20;
do
{
cout<<x;
x = x - 2;
} while (x > = 2);
```

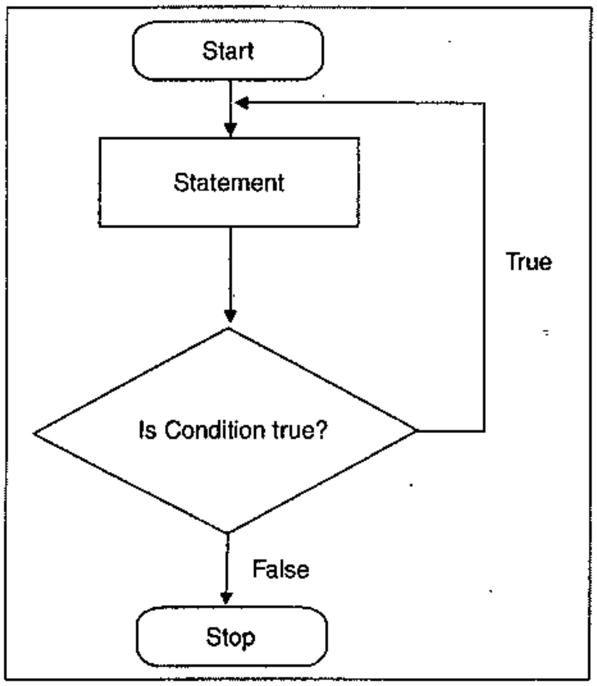


Fig. 8 *Do..while loop*

Example

```
/* Demo of  do...while statement */
#include <iostream.h
   void main()
   {
   int selection = 0;
   cout<<"You chose Menu Option \n"<<selection;
   do
   {
      cout<<"\n";
      cout<<"\n1 - Add a record";
      cout<<"\n2 - Change a record";
      cout<<"\n3 - Delete a record";
      cout<<"\n4 - Quit";
      cout<<"\n";
      cout<<"\nEnter a selection: " ;
      cin>>selection ;
   }  while ( selection < 1 || selection > 4 );
      }
```

The For, The While, or The Do...While

If you look at the syntax provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to

NOTES

what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you aren't dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best. Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.

Example

```
    #include <iostream.h>
    void main()
    {
        int sam;
        long lar;
        const int MAX=65535;
        cout << "Enter a small number: ";
        cin >> sam;
        cout << "Enter a large number: ";
        cin >> lar;
        cout << "small: " << sam<< "...";
        while (sam < lar && lar > 0 && sam < MAX)
        {
        if (sam % 5000 = = 0) // write a*atevery 5000 lines
        cout << " * ";
        sam++;
        lar = lar - 2;
        }
        cout << "\nSmall: " << sam << " Large: " << lar
<< endl;
    }
```

Infinite Loop

The condition you use for testing in a while loop can be any valid C++ expression. As long as that condition remains true, the while loop will continue. You can create a infinite loop that will never end by using the number 1 for the condition to be tested. Since 1 is always true, the loop will never end, unless a break statement is reached. like:

Example

```
    #include <iostream.h>
    void main()
    {
    int counter = 0;
       while (1)          //infinite loop
       {
          counter ++;
          if (counter > 25)
          break;          //break condition
       }
          cout << "Counter: " << counter << "\n";
    }
             Output: Counter: 26
```

While vs Do .. While Loop

It is possible that the body of a while loop will never execute. The while statement checks its condition before executing any of its statements, and if the condition evaluates false, the entire body of the while loop is skipped. The do...while loop executes the body of the loop before its condition is tested and ensures that the body always executes at least one time.

Example

```
       // Demonstrates do while
       #include <iostream.h>
       void main()
       {
       int counter;
       cout << "How many hellos in loop you want ? ";
       cin >> counter;
       do
       {
          cout << "Hello Friend \n";
          counter—;
       }  while (counter >0 );
       cout << "Value of Counter is: " << counter <<
endl;
    }
```

```
Output: How many hellos in loop you want ? 2
Hello
Hello
Value of Counter is: 0
```

In above program the user is prompted for enter a starting value, which is stored in the integer variable counter. In the do...while loop, the body of the loop is entered before the condition is tested, and therefore the body of the loop is guaranteed to run at least once even the condition can be false.

Switch Statement

For decision making based programs, you have seen the use of if and if/else statements. These can become quite confusing when if nested too deeply, but in C++ we have an alternative. Use switch statement, unlike if, which evaluates one value, switch statements allow you to branch on any of a number of different values. The general form of the switch statement is:

```
switch (condition / expression)
{
case One: statement;
            break;
case Two: statement;
            break;
case N:   statement;
            break;
default:    statement;
}
```

Example

```
// Demonstrates switch statement
#include <iostream.h>
void  main()
{
   int number;
   cout << "Enter a number between 1 and 7: ";
   cin >> number;
   switch (number)
{
        case 0:    cout << "Amrapali Institute
Haldwani -List of courses!";
```

```
            break;
        case 6:  cout << "B.Tech!\n";
        case 5:  cout << "MCA!\n";
        case 4:  cout << "MBA!\n";
        case 3:  cout << "BBA!\n";
        case 2:  cout << "BCA!\n";
        case 1:  cout << "BHMCT!\n";
        break;
        default: cout << "Please enter value 1-6!\n";
            break;
    }
      cout << "\n\n";
    }
    Output: Enter a number between 1 and 6:3
    BBA!
    BCA!
    BHMCT!
    Enter a number between 1 and 6: 8
    Please enter value 1-5!
```

BREAKING AND CONTROL STATEMENTS

Break and Continue

Any times in a loop if there is a need to return to the top of the loop before the entire set of statements in the loop is executed. The continue statement jumps back to the top of the loop. On the other hand, if you may want to exit the loop before the exit conditions are met. The break statement immediately exits the while loop, and program execution resumes after the closing brace.

```
Loop ( condition)         while (x<=10)
{                         {
statement1 ;              cout<<x;
if (condition)            if(x=5)
{                         {
continue ;                continue ;
statement2;               cout<<x+5;
}                         }
statement3 ;              cout<<"end of loop";
```

NOTES

```
}                               }
Loop ( condition )              while (x<=10)
{                               {
statement1 ;                          cout<<x;
if ( condition)                    if(x=5)
{                               {
break ;         ────────┐             break ;
statement2;             │          cout<<x+5;
}                       │                                      }
statement3 ;            ▼       cout<<"end of loop";
}
```

STUDENT ACTIVITY

1. What is Object Oriented model of programming?

2. What happens if you create a loop that never ends?

SUMMARY

- Object-oriented programming treats data as a critical element in the program development and does not allow it to flow freely around the system.
- Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate that data.
- Objects take up space in the memory. When a program is executed, the objects interact by sending messages to one another.
- A **structured object** is an object of a class with attributes, operations, and relationships.
- C++ is a extension of C, but it does not mean that you should learn C first. It is unnecessary to learn C first one can easily start C++ programming because C++ allow you to write code in C style.
- A constant is a data storage lochildion, constants don't change their values in a program.
- C++ statements always end with a semicolon (except for preprocessor directives such as #define and #include).

SELF ASSESSMENT QUESTIONS

1. What is Object-oriented programming? How is it different from the procedure-oriented programming?
2. Distinguish between the following terms:
   (*a*) Objects and classes
   (*b*) Data abstraction and data encapsulation
   (*c*) Inheritance and polymorphism
3. What is the difference between procedural vs object-oriented programming?
4. Explain the features of object oriented programming languages.
5. Why we should write reusable source codes and how C++ is helpful to write reusable codes?
6. What is a class and an object?
7. What is the difference between an integar variable and a floating-point variable?
8. What are the differences between an unsigned short int and a long int?

9. What are the advantages of using a symbolic constant rather than a literal constant?
10. What are the advantages of using the const keyword rather than #define?
11. What are rules for a good or bad variable name?
12. What is an expression?
13. Is x = 15/5 an expression? What is its value?
14. What is the value of 201/4? (if you will use integer)
15. What is the value of 201 % 4?
16. If myAge, a, and b are all int variables, what are their values after:

    myAge = 39;

    a = myAge++;

    b = ++myAge;
17. What is the value of 8+2*3?
18. What is the difference between x = 3 and x == 3?
19. Do the following values evaluate to TRUE or FALSE?

    (*a*) 0

    (*b*) 1

    (*c*) –1

    (*d*) x = 0
20. What is Conditional decision making, how it is important in logical design development in a program?
21. Is if (5 + 7 >11) is a valid if statement? What will be the return value of it?
22. What is nested if, describe with the help of an example?
23. What are common operators, used to design multiple conditions in if statement state some examples?
24. If myAge, *a*, and *b* are all int variables, what are their values after:

    myAge = 39;

    (*a*) = myAge++;

    (*b*) = ++myAge;
25. How do you initialize more than one variable in a for loop?
26. Why is goto avoided?
27. Is it possible to write a for loop with a body that is never executed?
28. Is it possible to nest while loops within for loops?
29. Is it possible to create a loop that never ends? Give an example.

SECTION B

NOTES

UNIT 2 FUNCTIONS, STRUCTURES, POINTERS AND UNIONS

★ LEARNING OBJECTIVES ★

- Defining a Function
- Types of Functions
- Call by Value and Call by Reference
- Preprocessor
- Header Files and Standard Functions
- Pointers and Structures
- Unions

DEFINING A FUNCTION

A function is a block of statements with a name. In your mobile set you have several functions like call a number, send a SMS or read a message. Any time you can use that without defining that, this is true with C++ functions. Once a function has been designed you can call it to perform your task. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function. A function is independent. A function can perform its task without interference from or interfering with other parts of the program. A function performs a specific task like send your photo to your girlfriend using MMS services of your handset. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root. A function can return a value to the calling program like your messege has been delivered. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

If you want to define a functions in C++, you should use the following steps:

(*a*) prototype the function

```
    //int sum ( int , int )
```

(*b*) define the function

```
    //int sum( int  x, int y )
  {
        z = x + y ;
        return z ;
  }
```

(*c*) use or call the functions

```
    // z=sum(a,b);
```

NOTES

Example

```
/* Demo of  function for sum   */
#include <iostream.h>
int sum(int , int );    // function prototype
int x , y , z ;
void main()
{
   cout<<"Enter two numbers ::";
   cin>>x>>y;
   z = sum (x, y);       //function call
   cout<<"The Sum of numbers is\n"<<z;
}
   // function definition
   int sum (int x, int y)
{
return x+y;
}
```

TYPES OF FUNCTIONS

There are three main types of function in C++

1. C style functions
2. Inline functions
3. Friend functions

NOTES

Inline Function

When you define a function, normally the compiler creates just one set of instructions in memory. When you call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If you call the function 5 times, your program jumps to the same set of instructions each time. This means there is only one copy of the function, not 5. There is some performance overhead in jumping in and out of functions. It turns out that some functions are very small, just a line or two of code, and some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions. The program runs faster if the function call can be avoided. If a function is declared with the keyword inline, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you have written the statements of the function right into the calling function. To declare a function inline use the keyword inline before the type of function.

Inline returntype functionname(passing parameter)

inline int double(int);

An inline function is a function whose code gets inserted into the caller's code stream. Like a #define macro, inline functions improve performance by avoiding the overhead of the call itself and (especially!) by the compiler being able to optimize through the call.

Example

```
      // inline function two compute square and cube of
integer numbers
      #include <iostream.h>
      inline double  square(int);
      inline double cube(int)
      void  main()
      {
         int num;
         double sq,cub;
         cout << "Enter a number";
         cin >> num;
         cout << "\n";
         sq = square(num);
         cout << "Square is :"<<sq << endl;
         cub = cube(num);
```

```
    cout << "Cube is : " << cub << endl;
  }
    double square(int num)
    {
       return num*num;
    }
    double cube(int num)
    {
    return num*num*num ;
    }
```

CALL BY VALUE AND CALL BY REFERENCE

Call by value

When you will use call by value style in a function you will pass actual variables to the function, and you can declare variables within the body of the function. This is done using value of variables, so named because they exist within the function itself.

The parameters passed into the function are real values of variables and can be used exactly as if they had been defined within the body of the function.

Program

```
#include <iostream.h>
     float Convert(float);
     int main()
     {
        float TempFer;
        float TempCel;
        cout << "Please enter the temperature in
Fahrenheit: ";
        cin >> TempFer;
        TempCel = Convert(TempFer);
        cout << "\nHere's the temperature in Celsius: ";
        cout << TempCel << endl;
           return 0;
     }
     float Convert(float TempFer)
```

```
{
    float TempCel;
    TempCel = ((TempFer - 32) * 5) / 9;
    return TempCel;
}
```

Call by Reference

When declaring a reference variable you must also make it refer to something at the same time. To declare on you simply make a variable of the type you are going to be referring to, make up your own name, and put an ampersand (&) in front of it:

```
int &ref;
```

That creates a reference variable called ref of type integer. Of course this doesn't do anything because you can't assign references to other variables at any other time than declaration. So to make ref refer to something we have to do it in the declaration:

```
int &ref = x;
```

This is all assuming we have a variable called x and that it is also an integer (int). But after doing this, anything we do to ref will effect x. Try this source code (cut and paste).

Program

```
#include <iostream.h>
void main()
{
    int x = 10;           // create integer variable
called x
    int &ref = x;         // make a reference variable
that refers to x
    cout  "x is "  x  " and ref is "  ref  endl;
    cout  "Now we change ref to equal 25 ... "  endl;
    ref = 25;
    cout  "And now x is "  x  " and ref is "  ref  endl;
}
```

Using Reference Variables in Functions

The previous section was an intro into reference variables. But to be quite honest if you use them like that then they're not really necessary. Reference variables come into shine when it comes to functions. The point of reference variables and functions is that you can pass a variable as a parameter and have the variable changed in the function. Like in the following code snippet.

Program

```
#include <iostream.h>
void times2(int &x);   // function prototype
void main()
{
int var;     // declare var as integer variable
var = 10; // put value of 10 in var
cout << "var is " << var  <<endl;
times2(var); // call 'times2()' with var as parameter
cout << "var is now " << var <<endl;
}
void times2(int &x)
{
x = x * 2;
}
```

With references you could get multiple values, like in the following.

Program

```
#include <iostream.h>
void times2(int &v1, int &v2);  // function prototype
void main()
{
   int x,y;
   x = 10;
   y = 15;
   cout << "x is "x" and y is "<<y  <<endl;
   times2(x,y);
   cout << "x is now "x" and y is now " << y << endl;
   }
   void times2(int &v1, int &v2)
   {
      v1 = v1 * 2;
      v2 = v2 * 2;
```

NOTES

PREPROCESSOR

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a pound sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. 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 (\).

#define

To define preprocessor macros we can use #define. Its format is:

```
#define identifier replacement
#define TRUE 1
```

When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of identifier by replacement.

```
#define TABLESIZE 100
int  table1[TABLESIZE];
int  table2[TABLESIZE];
```

After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:

```
int  table1[100];
int  table2[100];
```

Example

```
// Use of #define to  develop a function getmin() to
find minimum in two numbers
include <iostream.h>
#define getmin(a,b)  ((a)<(b)?(a):(b))
void main()
{
int x=5, y=8;
y= getmin(x,y);
cout << y << endl;
}
```

# AND # #

Function macro definitions accept two special operators (# and ##) in the replacement sequence. If the operator # is used before a parameter is used in the replacement sequence, that parameter is replaced by a string literal (as if it were enclosed between double quotes), The operator ## concatenates two arguments leaving no blank spaces between them. Like:

```
      #define str(x)   #x
      cout << str(test);
      output  will be "test"

      #define join(a,b)     a ## b
      join(c,out)<<"test"
      cutput will be cout<< "test"
```

#undef

If you have defined some value using #define you can erase it using #undef in a program. Like:

```
      #define TABLESIZE 100
      int  table1[TABLESIZE]
      #undef  TABLESIZE
      #define TABLESIZE 200
      int  table2[TABLESIZE]
```

You will get

```
         table1[100]
         table2[200]
```

HEADER FILES AND STANDARD FUNCTIONS

Every implementation of C++ includes the header files and standard library functions, and most include additional libraries as well. Libraries are sets of functions that can be linked into your code. You've already used a number of standard library functions and classes, most notably from the iostreams.h library.

To use a library, you typically include a header file in your source code, much as you did by writing #include <iostream.h> in many of the examples in this book. The angle brackets around the filename are a signal to the compiler to look in the directory where you keep the header files for your compiler's standard libraries. There are dozens of libraries, covering everything from file manipulation to setting the date and time to math functions.

NOTES

The main header files are :

```
Iostream.h
Conio.h
Math.h
Graphics.h
String.h
Iomanip.h
Time.h
Fstream.f
```

Using of time.h header file

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

     int main()
     {
        time_t  currentTime;
        // get and print the current time
        ctime (&currentTime); // fill now with the current
time
        cout << "It is now " << ctime(&currentTime) << endl;
        struct tm * ptm= localtime(&currentTime);
        cout << "Today is " << ((ptm->tm_mon)+1) << "/";
        cout << ptm->tm_mday << "/";
        cout << ptm->tm_year << endl;
        cout << "\nDone.";
     return 0;
     }
```

```
Output: It is now Mon Mar 31 13:50:10 1997
Today is 3/31/97
```

POINTERS AND STRUCTURES

To declare a pointer in a program, write the type of the variable or object whose address will be stored in the pointer, followed by the pointer operator (*) and the name of the pointer. For example,

```
     int * point = 0;
```

To assign or initialize a pointer, prepend the name of the variable whose address is being assigned with the address of operator (&). For example,

```
int theVar = 5;
      int * point = & theVar;
```

To dereference a pointer, prepend the pointer name with the dereference operator (*). For example,

```
int theValue = *point
```

you've seen step-by-step details of assigning a variable's address to a pointer. In practice, though, you would never do this. After all, why bother with a pointer when you already have a variable with access to that value? The only reason for this kind of pointer manipulation of an automatic variable is to demonstrate how pointers work. Pointers are used, most often, for following tasks:

- Managing data on the free store.
- Accessing class member data and functions.
- Passing variables by reference to functions.

Reference Variable

A reference is work as an alias for an object, when you create a reference, you initialize it with the name of another object. From that moment on, the reference acts as an alternative name for the target object, and anything you do to the reference is really done to the target object.

You can create a reference by writing the type of the target object, followed by the reference operator (&), followed by the name of the reference. References can use any legal variable name. If you have an integer variable named myint, you can make a reference to that variable by writing the following:

```
      int & myref = myint;
```

now myref will act as a reference variable for myint in the whole program.

Example

```
      // show the use of reference variables
      // Demo of References
      #include <iostream.h>
      void  main()
      {
         int aone;
         int &ref = aone;
```

NOTES

```
        aone = 15;
        cout << "AOne: " << aone << endl;
        cout << "ref: " << ref << endl;
        ref = 74;
        cout << "AOne: " << aone << endl;
        cout << "ref: " << ref << endl;
    }
```

Example

```
    //Use of  Address of Operator  - & - on References.
    #include <iostream.h>
    void main()
    {
        int intOne;
        int &SomeRef = intOne;
        intOne = 5;
        cout << "intOne: " << intOne << endl;
        cout << "SomeRef: " << SomeRef << endl;
        cout << "&intOne: "  << &intOne << endl;
        cout << "&SomeRef: " << &SomeRef << endl;
    }
```

Reference to Objects

You can create a reference to an object, but not to a class. You can not write this:

```
    int  & intref = int;     // wrong
```

You must initialize intref to a particular integer value, such as this:

```
    int bignumber = 1200;
    int & intref = bignumber;
```

In the same way, you don't initialize a reference to a class like:

```
    student  & rstu = student;   // wrong
```

You must initialize rstu to a particular student object first to set reference like:

```
    student nitinpadlia;
    student & rstu = nitinpadlia ;
```

Example

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

```
{
   public:
        SCat (int age, int weight);
        ~SCat() {}
   int GetAge() { return itsAge; }
   int GetWeight() { return itsWeight; }
private:
   int itsAge;
   int itsWeight;
};
SCat::SCat(int age, int weight)
{
   itsAge = age;
   itsWeight = weight;
}
void main()
{
   SCat juli (15,3);
   SCat & rCat = juli;
   cout << "juli is: ";
   cout << juli.GetAge() << " years old. \n";
   cout << "And juli's  weight is : ";
   cout << rCat.GetWeight() << " Kg. \n";
}
```

UNIONS

Unions are similar to structures. A union is declared and used in the same ways that a structure is. A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the same area of memory. They are laid on top of each other.

Defining, Declaring, and Initializing Unions

Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct. To define a simple union of a char variable and an integer variable, you would write the following:

NOTES

```
union shared
{
char c;
int i;
};
```

This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. Unlike a structure that would hold both values, the union can hold only one value at a time.

STUDENT ACTIVITY

1. Define Inline functions.

2. Name four preprocessor derivatives with examples.

NOTES

SUMMARY

- A function can return a value to the calling program like your messege has been delivered.
- An inline function is a function whose code gets inserted into the caller's code stream.
- The point of reference variables and functions is that you can pass a variable as a parameter and have the variable changed in the function.
- Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor.
- If you have defined some value using #define you can erase it using #undef in a program.
- A reference is work as an alias for an object, when you create a reference, you initialize it with the name of another object.
- Unions are similar to structures. A union is declared and used in the same ways that a structure is.

SELF ASSESSMENT QUESTIONS

1. What do you mean by preprocessor derivatives?
2. How do you use #define to set values of constants in your program?
3. What is the difference between #define debug 0 and #undef debug?

SECTION C

UNIT 3 CLASSES, INHERITANCE AND CONSTRUCTORS

NOTES

★ LEARNING OBJECTIVES ★

- Classes
- Member Functions
- Objects
- Array of Objects
- Constructors
- Copy Constructors
- Destructors
- Inline Member Functions
- Static Class Member-Functions
- Friend Functions
- Dynamic Memory Allocation
- Inheritance
- Virtual Base Class
- Abstract Classes
- Constructors in Derived Classes
- Nesting of Classes

CLASSES

Programs are usually written to solve real-world problems, such as keeping track of employee records in an organization like Amrapali Institute or simulating the workings of a heating system. Although it is possible to solve complex problems by using programs written with only integers and characters data types, but it is more easier to solve large, complex problems if you can create objects using base classes. In other words, simulating the workings of a heating system is easier if you can create variables that represent rooms, heat sensors, thermostats, and boilers. The closer these variables correspond to reality, the easier it is to write the program. To solve real world problems there is a need to create new data types also known as user defined data types, a class help us to design a new data type with attributes and operations or functions in a single unit. A class defines a data type, much like a struct C. In a computer science sense, a type consists of both a set of states and a set of operations which transition between those states. You can make a new data type by declaring a class. A class is just a collection of variables often of different types combined with a set of related functions. One way to think about a car is as a collection of wheels, doors, seats, windows, and so forth. Another way is to think about what a car can do: It can move, speed up, slow down, stop, park, and so on. A class enables you to encapsulate, or bundle, these various parts and various functions into one collection, which is called an object.

CAR class
wheels
doors
seats
windows
Functions
Move
Start
Stop
Park

Fig. 1 *a CAR class*

CAR c1, c2;

Two objects of CAR, having same attributes and functions

Encapsulating everything you know about a car into one class has a number of advantages for a programmer. Everything is now in one place, which makes it easy to refer to, copy, and manipulate the data using objects of class.

Example

A class employee with following attributes empno, empname, empdept, empsalary and functions join, computesalary, printdata, printsalary.

Class employee
Empno – integer
Empname – string
Empdept – string
Empsalary – integer

NOTES

Functions

Join()

Computesalary()

Printdata()

Printsalary()

Example

A class bankaccount with following attributes accno, custname, custadd, openamount, closingamount and functions open, deposit, withdrawal, checkbalance, close.

Class bankaccount

Accno – integer

Custname – string

Custadd – string

Openamount – integer

Closingamount – integer

Functions

Open()

Deposit()

Withdrawal()

Checkbalance()

Close()

MEMBER FUNCTIONS

class can consist of any combination of the variable types and also her class types. The variables in the class are referred to as the mber variables or data members. Like a Car class might have member iables representing the seats, windows, doors, tires etc. Member ables, also known as data members, are the variables in your class. ber variables are part of your class, just like the wheels and e are part of your car class. The functions in the class typically ılate the member variables. They are referred to as member ıs or methods of the class. Methods of the Car class might Start() and Stop(). Member functions, also known as methods, nctions in your class. Member functions are as much a part ss as the member variables. They determine what the objects s can do.

function

d
er function

ed programming languages like C++, "obje usually ıce of a class." Thus a class defines the b viour of

ct

NOTES

possibly many objects. You can say now an object is an individual instance of a class. You can define an object of your new clas t as you define an integer variable. Like:

```
Car c1,                    // One object of Ca
Employee e1,e2,            // Two objects
Student  s1,s2,s3          // Three ob
```

CAR

NOTES

```
{
public:             //members which follow are public
    int wheels;
    int doors;
};
void main()                 //Object     member variable
{
    Car maruti;
    maruti.wheels = 4; //ass which has";
    cout << "Maruti  is << " Wheels\n";
    cout << maruti.
}
```

Example //add and subtract two numbers using classes and objects

```
#include <iostream.h>
class addsub
{
 int x;
 int y; //private data members
 public:                          // public member
 void twosum();                   // class defi
 void twosub();                   // define me
};
void addsub::twosum()
{
 cin>>x;
 cin>>y;
 int z=x+y;
 cout<< "Sum is ="<<z;
}
void addsub::twosub()
{
 cin>>x;
 cin>>y;
 int z=x-y;
```

De
the
and

Functions

Join()

Computesalary()

Printdata()

Printsalary()

Example

A class bankaccount with following attributes accno, custname, custadd, openamount, closingamount and functions open, deposit, withdrawal, checkbalance, close.

Class bankaccount

Accno – integer

Custname – string

Custadd – string

Openamount – integer

Closingamount – integer

Functions

Open()

Deposit()

Withdrawal()

Checkbalance()

Close()

MEMBER FUNCTIONS

A class can consist of any combination of the variable types and also other class types. The variables in the class are referred to as the member variables or data members. Like a Car class might have member variables representing the seats, windows, doors, tires etc. Member variables, also known as data members, are the variables in your class. Member variables are part of your class, just like the wheels and engine are part of your car class. The functions in the class typically manipulate the member variables. They are referred to as member functions or methods of the class. Methods of the Car class might include Start() and Stop(). Member functions, also known as methods, are the functions in your class. Member functions are as much a part of your class as the member variables. They determine what the objects of your class can do.

OBJECTS

In object oriented programming languages like C++, "object" usually means "an instance of a class." Thus a class defines the behaviour of possibly many objects. You can say now an object is an individual instance of a class. You can define an object of your new class just as you define an integer variable. Like:

```
Car c1,               // One object of Car class
Employee e1,e2,       // Two objects of Employee class
Student s1,s2,s3      // Three objects of student class
```

CAR class

wheels

doors

seats

windows

Functions

Move

Start

Stop

Park

CAR	**c1;**
Class	Object
Data members	**Functions**
Wheels	Move
Doors	Start
Seats	Stop
Windows	Park

Class Declaration

To declare a class, use the class keyword followed by an opening brace, and then list the data members and methods of that class. End the declaration with a closing brace and a semicolon. Here's the declaration of a class called Car:

```
class Car
{
int doors;                 //member variable
int wheels;
Start();                   //member function
Stop();
};
```

Declaring this class doesn't allocate memory for a Car. It just tells the compiler what a Car is, what data it contains (wheels and doors), and what it can Start() and Stop(). It also tells the compiler the size of a Car in bytes, to know how much bytes the compiler must set aside for each Car object that you will create in future. In this example, if an integer is two bytes, a Car is only four bytes big: doors is two bytes, and wheels is another two bytes. Start() and Stop() takes up no bytes, because no storage space is reserved for member functions.

Public Vs Private

Some more keywords are used in the declaration of a class. Two of the most important are public and private. All members of a class data and methods are private by default. Private members can be accessed only within methods of the class itself. Public members can be accessed through any object of the class. This distinction is both important and confusing. To make it a bit clearer, consider an example from earlier in this chapter:

```
class Car
{
int doors;
int wheels;
Start();
Stop();
};
```

in this declaration, doors, wheels, Start() and Stop() are all private, because all members of a class are private by default. This means that unless you specify otherwise, they are private. However, if you now declare object of Car class like:

```
Car c1;
c1.wheels=6; // error! can't access private data!
```

the compiler will show this as an error. Because you cannot access private data values uirectly.

Now change the class declaration as:

```
class Car
{
public:
int doors;
int wheels;
Start();
Stop();
};
```

Example

```
#include <iostream.h>
class Car    // declare the class object
```

NOTES

```
{
public:     //-members which follow are public
   int wheels;
   int doors;
};
void main()
{
   Car maruti;         //Object declaration
   maruti.wheels = 4; //assign to the member variable
   cout << "Maruti  is a car which has";
   cout << maruti.wheels << " Wheels\n";
}
```

Example

```
    //add and subtract two numbers using classes and
objects
    #include <iostream.h>
    class addsub
    {
    int x;
    int y; //private data members
    public:
    void twosum();           // public member function
    void twosub();
    };                       // class defined
    void addsub::twosum()    // define member function
    {
    cin>>x;
    cin>>y;
    int z=x+y;
    cout<< "Sum is ="<<z;
    }
    void addsub::twosub()
    {
    cin>>x;
    cin>>y;
    int z=x-y;
    cout<< "Subtraction is ="<<z;
    }
    void main()
    {
    addsub s;    // object declaration of addsub class
    s.twosum();  // calling member function with object
    s.twosub();
    }
```

Example

```
    //multiply and divide two numbers using classes and
objects
    #include <iostream.h>
    class addsub
    {
    int x;
    int y;          // private data members
    public:
    void twomult()  // public member function inside class
    {
    cin>>x;
    cin>>y;
    int z=x*y;
    cout<< "Product is ="<<z;
    }
    void twodiv()
    {
    cin>>x;
    cin>>y;
    int z=x-y;
    cout<< "Division is ="<<z;
    }
    };        // class defined
    void main()
    {
    addsub s;              //object declaration of addsub
class
s.twomult(); // calling member function with object
s.twodiv();
}
```

NOTES

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. The class must have a default constructor that takes no arguments so that the objects can be created when the array is defined.

☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺

Fig. 2 *An array of faces*

S A C H I N T E N D U L K A R

Fig. 3 *An array of characters*

An array is a collection of similar data values in a single unit.

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 operator (.) to access the particular member variable. Like:

- int x[10]; // array of 10 integer data types
- float y[12]; // array of 12 float data types
- child c[5]; // array of 5 child class objects.

Example

```
// Demo  - An array of objects
#include <iostream.h>
class CHILD
{
   public:
      CHILD() { itsAge = 1; itsWeight=5; }
      ~CHILD() {}
   int GetAge() const { return itsAge; }
   int GetWeight() const { return itsWeight; }
   void SetAge(int age) { itsAge = age; }
private:
        int itsAge;
        int itsWeight;
  };
  void  main()
  {
      CHILD suhani[5];     // array of objects of CHILD
class
      int i;
      for (i = 0; i < 5; i++)
         suhani[i].SetAge(2*i  +1);
   for (i = 0; i < 5; i++)
  {
      cout << "Child #" << i+1<< ": ";
      cout << suhani[i].GetAge() << endl;
  }
  }
```

Array of Pointers

The arrays of objects usually store all their members in a stack. Usually stack memory is severely limited, whereas free store memory is far larger. It is possible to declare each object on the free store and then to store only a pointer to the object in the array. This dramatically reduces the amount of stack memory used and fasten the processing speed. As an indiaimcaion of the greater memory that this enables, the array in next example extended from 5 to 500.

Example

```
// demo of An array of pointers to objects
#include <iostream.h>
class AIMCA
{
   public:
       AIMCA() { itsAge = 1; itsWeight=5; }
       ~AIMCA() {}        // destructor
       int GetAge() const { return itsAge; }
       int GetWeight() const { return itsWeight; }
       void SetAge(int age) { itsAge = age; }
  private:
       int itsAge;
```

NOTES

```
        int itsWeight;
    };
          void main()
          {
        AIMCA * Family[500];
        int i;
        AIMCA * pAimca;
        for (i = 0; i < 500; i++)
        {
            pAimca = new AIMCA;
            pAimca->SetAge(2*i  +1);
            Family[i] = pAimca;
        }
            for (i = 0; i < 500; i++)
        {
        cout << "Aimca #" << i+1 << ": ";
        cout << Family[i]->GetAge() << endl;
    }
}
```

CONSTRUCTORS

There are two ways to define an integer variable. You can define the variable and then assign a value to it later in the program. For example,

```
int Weight;     // define a variable
...             // other code here
Weight = 7;     // assign it a value
```

Or you can define the integer and immediately initialize it. For example,

```
int Weight = 7;    // define and initialize to 7
```

Initialization combines the definition of the variable with its initial assignment. Nothing stops you from changing that value later. Initialization ensures that your variable is never without a meaningful value. How do you initialize the member data of a class? Classes have a special member function called a constructor. The constructor can take parameters as needed, but it cannot have a return value—not even void. The constructor is a class method with the same name as the class itself.

Whenever you declare a constructor, you'll also want to declare a destructor. Just as constructors create and initialize objects of your class, destructors clean up after your object and free any memory you might have allocated. A destructor always has the name of the class, preceded by a tilde (~). Destructors take no arguments and have no return value. Therefore, the Cat declaration includes

```
~Cat();
```

Default Constructors and Destructors

If you don't declare a constructor or a destructor, the compiler makes one for you. The default constructor and destructor take no arguments and do nothing. What good is a constructor that does nothing? In part, it is a matter of form. All objects must be constructed and destructed, and these do-nothing functions are called at the right time. However, to declare an object without passing in parameters, such as

```
Cat Reena;      // Rags gets no parameters
```

you must have a constructor in the form

```
Cat();
```

When you define an object of a class, the constructor is called. If the Cat constructor took two parameters, you might define a Cat object by writing

```
Cat Reena(5,7);   // Parameterized constructor
```

If the constructor took one parameter, you would write

```
Cat Reena(3);
```

In the event that the constructor takes no parameters at all, you leave off the parentheses and write

```
Cat Frisky;
```

This is an exception to the rule that states all functions require parentheses, even if they take no parameters. This is why you are able to write

```
Cat Reena;
```

which is a call to the default constructor. It provides no parameters, and it leaves off the parentheses. You don't have to use the compiler-provided default constructor. You are always free to write your own constructor with no parameters. Even constructors with no parameters can have a function body in which they initialize their objects or do other work. As a matter of form, if you declare a constructor, be sure to declare a destructor, even if your destructor does nothing. Although it is true that the default destructor would work correctly, it doesn't hurt to declare your own. It makes your code clearer.

Now rewrite the Cat class to use a constructor to initialize the Cat object, setting its age to whatever initial age you provide, and it demonstrates where the destructor is called.

Program

```
    #include <iostream.h>    // for cout
    class Cat  // begin declaration of the class
```

```
private:
    // one data field: ptr to allocated string
    char *str;
};
```

Concerning this interface we remark the following:

The class contains a pointer char *str, possibly pointing to allocated memory. Consequently, the class needs a constructor and a destructor. A typical action of the constructor would be to set the str pointer to 0. A typical action of the destructor would be to release the allocated memory. For the same reason the class has an overloaded assignment operator. The code of this function would look like:

String const & String::operator=(String const & other)

```
{
    if (this != & other)
    {
        delete str;
        str = strdupnew(other.str);
    }
    return (*this);
}
```

The class has, besides a default constructor, a constructor which expects one string argument. Typically this argument would be used to set the string to a given value, as in:

```
String a("Hello World!\n");
```

The only interface functions are to set the string part of the object and to retrieve it. let's consider the following code fragment. The statement references are discussed following the example:

```
String a  ("Hello World\n"),  b, c = a;
int main()
{
    b = c;
    return (0);
}
```

Statement 1. This statement shows an initialization. The object a is initialized with a string "Hello World". This construction of the object a therefore uses the constructor which expects one string argument. It should be noted here that this form is identical to

```
String     a = "Hello World\n";
```

Even though this piece of code uses the operator =, this is no assignment: rather, it is an initialization, and hence, it's done at construction time by a constructor of the class String.

Statement 2. Here a second String object is created. Again a constructor is called. As no special arguments are present, the default constructor is used.

Statement 3. Again a new object c is created. A constructor is therefore called once more. The new object is also initialized. This time with a copy of the data of object a.

This form of initializations has not yet been discussed. As we can rewrite this statement in the form

```
String    c(a);
```

it suggests that a constructor is called, with as argument a (reference to a) String object. Such constructors are quite common in C++ and are called copy constructors. More properties of these constructors are discussed below.

Statement 4. Here one object is assigned to another. No object is created in this statement. Hence, this is just an assignment, using the overloaded assignment operator.

The simple rule emanating from these examples is that whenever an object is created, a constructor is needed. All constructors have the following characteristics:

- Constructors have no return values.
- Constructors are defined in functions having the same names as the class to which they belong.

Therefore, we conclude that, given the above statement (3), the class String must be rewritten to define a copy constructor:

```
// class definition
class String
{
public:
    String(String const & other);
    };
// constructor definition
String::String(String const & other)
{
    str = strdupnew(other.str);
}
```

need them is when you use dynamic memory allocation, mess with things that need to be set back when your done, etc.

NOTES

To declare a destructor function is similar to declaring a constructor function. The destructor's name should be exactly the same as the name of the class (like a constructor), however it should also be preceded by a tilde (~). So for our class Cat the destructor prototype would be:

```
~Cat();
```

The biggest difference between constructors and destructors is that the latter cannot have any parameters.

Program

```
      #include <iostream.h>
      int num_date_objects;    // global variable to
keep track of the number     // of 'date' objects
      class date
      {
      public:
         // constructor!
         date(int y, int m, int d)
         {
            year = y;
            month= m;
            day  = d;
num_date_objects++; // add one to the number of date
objects, this
         // number will be THIS object's id number
         id = num_date_objects;
         cout << "Calling constructor, creating date object
#"  id  "!"   << endl;
         }
         // destructor!
         ~date()
         {
         cout << "Calling destructor!  *AWOOGA* *AWOOGA*!
date object #"
             <<  id  <<" has perished!"  <<endl;
         }
         int year, month, day, id;
};
void main()
{
   num_date_objects = 0;
   date neil_dob(1979,8,19);
   date joey_dob(1976,11,28);
}
```

INLINE MEMBER FUNCTIONS

The way to implement inline functions leaves a class interface intact, but mentions the keyword inline in the function definition. The interface and implementation in this case are as follows:

```
class Person
{
   public:
       ...
       char const *getname(void) const;
       ...
};
inline char const *Person::getname() const
{
   return (name);
}
```

Again, the compiler will insert the code of the function getname() instead of generating a call. However, the inline function must still appear in the same file as the class interface, and cannot be compiled to be stored in, *e.g.*, a library. The reason for this is that the compiler rather than the linker must be able to insert the code of the function in a source text offered for compilation. Code stored in a library is inaccessible to the compiler. Consequently, inline functions are always defined together with the class interface.

When to use inline functions

When should inline functions be used, and when not? There is a number of simple rules of thumb which may be followed:

Defining inline functions can be considered once a fully developed and tested program runs too slowly and shows 'bottlenecks' in certain functions. A profiler, which runs a program and determines where most of the time is spent, is necessary for such optimization. Inline

NOTES

```
    }
    class B        // class B: tries to touch
    {              // A's private parts
       public:
           void touch(A &a)
               { a.value++; }
    };
```

This code will not compile, since the classless function decrement() and the function touch() of the class B attempt to access a private datamember of A. We can explicitly allow decrement() to access A's data, and we can explicitly allow the class B to access these data. To accomplish this, the offending classless function decrement() and the class B are declared to be friends of A:

```
    class A
    {
       public:
           friend class B;    // B's my buddy, I trust him
           friend void decrement(A      // decrement() is
also a good pal
               &what);
           ...
    };
```

Friendship is not mutual by default. This means that once B is declared as a friend of A, this does not give A the right to access B's private members. Friendship, when applied to program design, is an escape mechanism which circumvents the principle of data hiding. Using friend classes should therefore be minimized to those cases where it is absolutely essential.

If friends are used, realize that the implementation of classes or functions that are friends to other classes become implementation dependent on these classes. In the above example: once the internal organization of the data of the class A changes, all its friends must be recompiled (and possibly modified) as well.

DYNAMIC MEMORY ALLOCATION

In C++ you can use two keywords **new** and **delete** for dynamic memory allocation.

Use of new

To allocate memory for objects or variables you can use the new keyword. New is followed by the type of the object that you want to allocate so that the compiler knows how much memory is required. Therefore, new unsigned short int allocates two bytes in the free store, and new long allocates four bytes. The return value from new is a memory address. It must be assigned to a pointer. To create an unsigned short on the free store, you might write

```
      unsigned short int * myp;
      myp = new unsigned short int;
```

You can, of course, initialize the pointer at its creation with

```
      unsigned short int * myp=new unsigned short int;
```

In either case, myp now points to an unsigned short int on the free store. You can use this like any other pointer to a variable and assign a value into that area of memory by writing

```
      *myp = 56;
```

This means, "allocate 56 at the value in myp," or "Assign the value 56 to the area on the free store to which myp points."If new cannot create memory on the free store (memory is, after all, a limited resource) it returns the null pointer. You must check your pointer for null each time you request new memory.

Use of delete

When you are finished with your area of memory, you must call delete on the pointer. Delete returns the memory to the free store. Remember that the pointer itself—as opposed to the memory to which it points—is a local variable. When the function in which it is declared returns, that pointer goes out of scope and is lost. The memory allocated with new is not freed automatically, however. That memory becomes unavailable—a situation called a memory leak. It's called a memory leak because that memory can't be recovered until the program ends. It is as though the memory has leaked out of your computer. To restore the memory to the free store, you use the keyword delete. For example,

```
      delete myp;
```

Program

```
      //Creating and deleting objects using new and delete.
      #include <iostream.h>
      class mycat
      {
      public:
```

NOTES

```
        mycat();
        ~mycat();
    private:
        int Age;
    };
    mycat::mycat()
    {
        cout << "Constructor called.\n";
        Age = 1;
    }
    mycat::~mycat()
    {
        cout << "Destructor called.\n";
    }
    void  main()
    {
        cout << "mycat  juli  \n";
        mycat  juli;
        cout << "mycat  *pcat = new mycat \n";
        mycat  * pcat = new  mycat;
        cout << "delete  pcat ..\n";
        delete pcat ;
        cout << "check where is juli \n";
    }
```

INHERITANCE

When you create a class and uses objects to work with class, with a set of attributes and functions, you have created something that is ready to pass these qualities on to it's children or subclass for reuse the main class to save your time and efforts on coding. This is called inheritance, every super class (parent) gives its qualities to its subclass (child). Inheritance in programs made possible to reuse the attributes and functions of a parent class into a child class.

The Family Inheritance

With all family trees we inherit the characteristics of our parents, grand parents and great grand parents. We can inherit that beautiful nose from our mothers side of the family, the buck teeth from our father, the long black hair from our great grandfather etc.

Inheriting Functions and Attributes

The functions and attributes of a class are the combination of two things, its own functions and attributes and the functions r and attributes of all its super classes. A class which adds new functionality to an existing class is said to derive or inherited from that original class. The original class is said to be the new class's base class.

Benefits of Inheritance

- You can reuse your base class functions and attributes in child class, without redifne or retype.
- You can merge functions of multiple classes in a single class, and you will get a new mixed class.
- Your new mobile handset carry many functions of your old one, thus we have to add just few new functions in old handset class.
- In case of windows operating system, all OS uses base classes as inherit class.

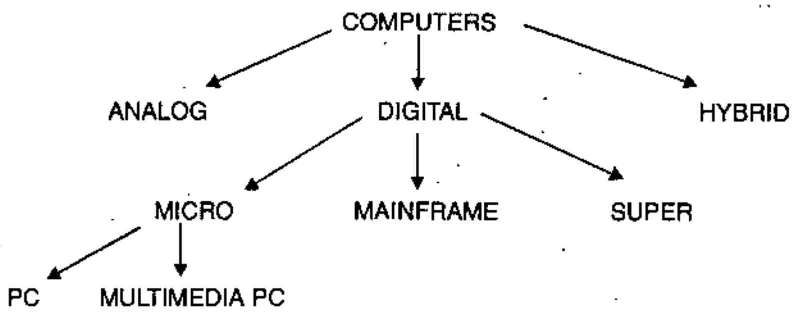


Fig. 4 *Example of Inheritance*

Now from figure you can say a PC inherit the features of MICRO computer, while a MICRO computer inherits the features of a DIGITAL computer and the parent class for all is COMPUTERS.

Types of Inheritance

You can design four type of Inheritance in C++

1. Single level (Parent – Child)
2. Multilevel (Grandparent – Parent – Child)
3. Multiple (Many parents – one child)
4. Hybrid (Mixture of multiple and multilevel)

NOTES

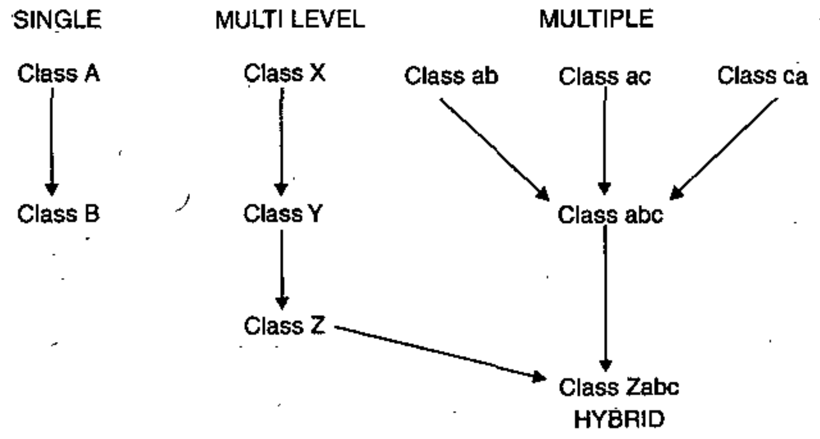


Fig. 5 *Types of inheritance*

Single Level Inheritance

To use single level inheritance in a program you should design a base class or parent class and then child class will inherit it. When you declare a class, *you can indicate what class it derives from by writing* a colon after the class name, the type of derivation (public or other), and the class from which it derives like:

```
Class child : public parent // syntax
Class MCA : public amrapali // example
```

The class from which you derive must have been declared earlier, or you will get a compiler error.

Program

```
//Creation of parent class
class institute
{
char name[25];
int telno;
public :
void getdata()
void showdata();
};
   //Creation of child class
   class student : public institute
{
int rollno;
char sname[25];
public :
    void readdata();
    void displaydata();
    };
    void institute::getdata()  //member  function  of
parent class
    {
    cin>>name;
    cin>>telno;
    }
    void student  ::readdata()      //member function of
child  class
    {
    cin>>sname;
    cin>>rollno;
    }
    void institute::showdata()    //member  function  of
parent class
    {
    cout<<name;
    cout<<telno;
    }
    void student::displaydata()    //member  function  of
child class
    {
    cout<<sname;
    cout<<rollno;
    }
  // Creation of class objects and function calling
void main
    {
    student s;
    s.getdata()  //function of parent class used by child
class object
    s.readdata();
    s.showdata(); //function  of  parent  class  used  by
child class object
s.displaydata();
    }
```

NOTES

Access Specifiers

There are, in total, three access specifiers:

- public
- protected
- private

All three can be used by a derived class. If a function has an object of your class, it can access all the public member data and functions. The member functions, in turn, can access all private data members and functions of their own class, and all protected data members and functions of any class from which they derive. However, private members are not available to derived classes. Protected data members and functions are fully visible to derived classes, but are otherwise private.

Visibility Modes

Accessible from $\longrightarrow$
$\downarrow$

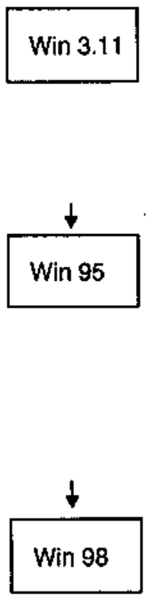
Access Mode	*Base Class*	*Derived Class*	*Outside the Class*
Public	Y	Y	Y
Private	Y	N	N
Protected	Y	Y	N

Fig. 6 *Table for visibility modes*

Multilevel Inheritance

In some situations classes can be derived more than one level, and we can form of a chain of classes derived by each others.

```
Class win3.11
{

}

class win95:public win3.11
{

}

class win98:public win95
{

}
```



```
class win2000:public win98
{

}
```

Fig. 7 *Multilevel inheritance in Windows OS*

NOTES

example and figure shows multilevel inheritance, in that win2000 derived from win98, and win98 is derived from win95 and the parent for all is win3.11 class. That shows win2000 will get the features of earlier parent classes.

Multiple Inheritance

The most common inheritance consists of an object deriving its foundation from another object. This is referred to as single inheritance. C++ allows an object to be based on more than one object. This is called refered to as multiple inheritance. When a class inherits properties or features of more than one base classes, it is known as multiple inheritance. Like:

```
Class a
{
int x,y;
public :
start();
sum()
getdata()
}
class b
{
int r,l;
public:
move();
setdata();
}
class c
{
float area;
public;
show();
```

NOTES

```
    stop();
    }
    class d :public a , public b , private c   //multiple
inheritance
    {
    start();
    move();
    stop();
    final();
    }
```

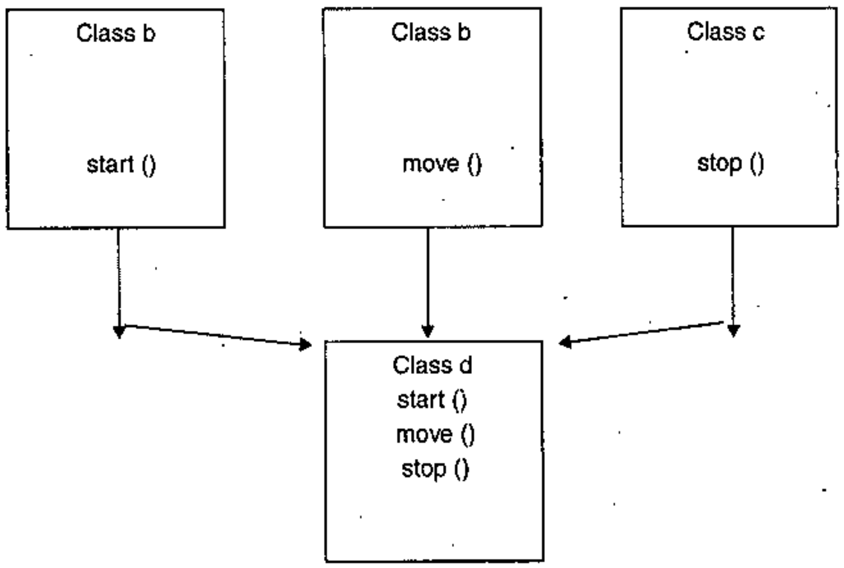


Fig. 8 *Multiple inheritance*

VIRTUAL BASE CLASS

```
class Truck: public Auto
{
  public:
     // constructors
     Truck();
     Truck(int engine_wt, int sp, char const *nm,
     int trailer_wt);
     // interface: to set two weight fields
     void setweight(int engine_wt, int trailer_wt);
     // and to return combined weight
```

```
        int getweight() const;
    private:
        // data
        int trailer_weight;
};
// example of constructor
Truck::Truck(int engine_wt, int sp, char const *nm,
int trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    trailer_weight = trailer_wt;
}
// example of interface function
int Truck::getweight() const
{
    return
    (           // sum of:
        Auto::getweight() +  // engine part plus
        trailer_wt           // the trailer
    );
}
```

ABSTRACT CLASSES

In *object-oriented programming,* **classes** are used to group related variables and functions. A class describes a collection of *encapsulated instance variables* and *methods* (functions), possibly with implementation of those types together with a constructor function that can be used to create objects of the class.

An **abstract class**, or *abstract base class* (ABC), is one that is designed *only* as a *parent class* and from which *child classes* may be derived, and which is not itself suitable for *instantiation*. Abstract classes are often used to represent *abstract* concepts or entities. The incomplete features of the abstract class are then shared by a group of sibling subclasses which add different variations of the missing pieces. In C++, an abstract class is defined as a class having at least one pure virtual method, *i.e.,* an *abstract method,* which may or may not possess an implementation.

Abstract classes are superclasses which contain *abstract methods* and are defined such that concrete subclasses are to extend them by implementing the *methods*. The *behaviours* defined by such a class are *"generic"* and much of the class will be *undefined* and unimplemented. Before a class derived from an abstract class can be instantiated, it must implement particular methods for all the abstract methods of its parent classes.

CONSTRUCTORS IN DERIVED CLASSES

When a derived class object is created, his base constructor is called first, creating a parent. Then the derived class constructor is called, completing the construction of the derived class object. When derived class object is destroyed, first the derived class destructor will be called and then the destructor for the parent class will be called. Each destructor is given an opportunity to clean up after its own part of derived class object.

Constructor of Base/parent class

Constructor of Derived /child class

...

....

..

Destructor of Derived/child class

Destructor of Base/parent class

NESTING OF CLASSES

Classes can be defined inside other classes. Classes that are defined inside other classes are called nested classes. A class can be nested in every part of the surrounding class: in the public, protected or private section. Such a nested class can be considered a member of the surrounding class. The normal access and visibility rules in classes apply to nested classes. If a class is nested in the public section of a class, it is visible outside the surrounding class. If it is nested in the protected section it is visible in subclasses, derived from the surrounding class, if it is nested in the private section, it is only visible for the members of the surrounding class. The surrounding class has no privileges with respect to the nested class. So, the nested class still has full control over the accessibility of its members by the surrounding class.

For example, consider the following class definition:

```
class Surround
{
    public:
        class FirstWithin
        {
            public:
                FirstWithin();
                int getVar() const
                {
                    return (variable);
                }
            private:
                int variable;
        };
    private:
        class SecondWithin
        {
            public:
                SecondWithin();
                int getVar() const
                {
                    return (variable);
                }
            private:
                int variable;
        };
        // other private members of Surround
};
```

In this definition access to the members is defined as follows:

The class FirstWithin is visible both outside and inside Surround. The class FirstWithin has therefore global scope. The constructor FirstWithin() and the memberfunction getVar() of the class FirstWithin are also globally visible. The int variable datamember is only visible for the members of the class FirstWithin. Neither the members of Surround nor the members of SecondWithin can access the variable of the class FirstWithin directly. The class SecondWithin is visible only inside Surround. The public members of the class SecondWithin can also be used by the members of the class FirstWithin, as nested classes can be considered members of their surrounding class. The constructor SecondWithin() and the memberfunction getVar() of the class SecondWithin can also only be reached by the members of Surround (and by the members of its nested classes).

The int variable datamember of the class SecondWithin is only visible for the members of the class SecondWithin. Neither the members of Surround nor the members of FirstWithin can access the variable of the class SecondWithin directly. If the surrounding class should have access rights to the private members of its nested classes or if nested classes should have access rights to the private members of the surrounding class, the classes can be defined as friend classes.

The nested classes can be considered members of the surrounding class, but the members of nested classes are not members of the surrounding class. So, a member of the class Surround may not access FirstWithin::getVar() directly. This is understandable considering the fact that a Surround object is not also a FirstWithin or SecondWithin object. The nested classes are only available as typenames. They do not imply containment as objects by the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then a pointer to a nested class object or a nested class datamember must be defined in the surrounding class, which can thereupon be used by the members of the surrounding class to access members of the nested class.

For example, in the following class definition there is a surrounding class Outer and a nested class Inner. The class Outer contains a memberfunction caller() which uses the inner object that is composed in Outer to call the infunction() memberfunction of Inner:

```
class Outer
{
    public:
        void caller()
        {
            inner.infunction();
        }
    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner  inner;
    };
```

Also note that the function Inner::infunction() can be called as part of the inline definition of Outer::caller(), even though the definition of the class Inner is yet to be seen by the compiler.

Inline functions can be defined as if they were functions that were defined outside of the class definition: if the function Outer::caller() would have been defined outside of the class Outer, the full class definition (including the definition of the class Inner would have been available to the compiler. In that situation the function is perfectly compilable. Inline functions can be compiled accordingly and there is, *e.g.*, no need to define a special private section in Outer in which the class Inner is defined before defining the inline function caller().

Defining Nested Class Members

Member functions of nested classes may be defined as inline functions. However, they can also be defined outside of their surrounding class. Consider the constructor of the class FirstWithin in the example of the previous section. The constructor FirstWithin() is defined in the class FirstWithin, which is, in turn, defined within the class Surround. Consequently, the class scopes of the two classes must be used to define the constructor. E.g.,

```
Surround::FirstWithin::FirstWithin()
{
    variable = 0;
}
```

Static (data) members can be defined accordingly. If the class FirstWithin would have a static unsigned datamember epoch, it could be initialized as follows:

```
Surround::FirstWithin::epoch = 1970;
```

Furthermore, both class scopes are needed to refer to public static members in code outside of the surrounding class:

```
void showEpoch()
{
    cout << Surround::FirstWithin::epoch = 1970;
}
```

Of course, inside the members of the class Surround only the FirstWithin:: scope needs to be mentioned, and inside the members of the class FirstWithin there is no need to refer explicitly to the scope. What about the members of the class SecondWithin? The classes FirstWithin and SecondWithin are both nested within Surround, and can be considered

NOTES

members of the surrounding class. Since members of a class may directy refer to each other, members of the class SecondWithin can refer to (public) members of the class FirstWithin. Consequently, members of the class SecondWithin could refer to the epoch member of FirstWithin as

```
        FirstWithin::epoch
```

Declaring Nested Classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers to objects of the other nested classes. For example, the following class Outer contains two nested classes Inner1 and Inner2. The class Inner1 contains a pointer to Inner2 objects, and Inner2 contains a pointer to Inner1 objects. Such cross references require forward declarations:

```
    class Outer
    {
        ...
        private:
            class Inner2;      // forward declaration
            class Inner1
             {
                 ...
                 private:
                     Inner2
                       *pi2;   // points to Inner2 objects
             };
             class Inner2
             {
                 ...
                 private:
                     Inner1
                       *pi1;   // points to Inner1 objects
             };
             ...
    };
```

Access to Private Members in Nested Classes

In order to allow nested classes to access the private members of the surrounding class or to access the private members of other nested classes or to allow the surrounding class to access the private members of nested classes, the friend keyword must be used. Consider the following situation, in which a class Surround has two nested classes FirstWithin and SecondWithin, while each class has a static data member int variable:

```
class Surround
{
    public:
        class FirstWithin
        {
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        int getValue();
    private:
        class SecondWithin
        {
            public:
                int getValue();
            private:
                static int variable;
        };
        static int variable;
};
```

If the class Surround should be able to access the private members of FirstWithin and SecondWithin, these latter two classes must declare Surround to be their friend. The function Surround::getValue() can thereupon access the private members of the nested classes. For example, (note the friend declarations in the two nested classes):

```
class Surround
{
    public:
        class FirstWithin
        {
            friend class Surround;
```

NOTES

```
                    public:
                        int getValue();
                    private:
                        static int
                            variable;
                };
                int getValue()
                {
                    FirstWithin::variable = SecondWithin::variable;
                    return (variable);
                }
            private:
                class SecondWithin
                {
                    friend class Surround;
                    public:
                        int getValue();
                    private:
                        static int
                            variable;
                };
                static int
                    variable;
        };
```

Now, in order to allow the nested classes to access the private members of the surrounding class, the class Surround must declare the nested classes as friends. The friend keyword may only be used when the class that is to become a friend is already known as a class by the compiler, so either a forward declaration of the nested classes is required, which is followed by the friend declaration, or the friend declaration. follows the definition of the nested classes. The forward declaration followed by the friend declaration looks like this:

```
    class Surround
    {
        class FirstWithin;
        class SecondWithin;
        friend class FirstWithin;
        friend class SecondWithin;
        public:
            class FirstWithin
            ... (etc)
```

Alternatively, the friend declaration may follow the definition of the classes. Note that a class can be declared a friend following its definition, while the inline code in the definition already uses the fact that it will be declared a friend of the outer class. Also note that the inline code of the nested class uses members of the surrounding class which have not yet been seen by the compiler. Finally note that the variable that is defined in the class Surround is accessed in the nested classes as Surround::variable:

```
    class Surround
    {
        public:
            class FirstWithin
            {
                friend class Surround;
               public:
                   int getValue()
                   {
                       Surround::variable = 4;
                       return (variable);
                   }
               private:
                   static int
                       variable;
            };
            friend class FirstWithin;
            int getValue()
            {
          FirstWithin::variable = SecondWithin::variable;
                return (variable);
            }
        private:
            class SecondWithin
            {
                friend class Surround;
               public:
```

NOTES

```
                int getValue()
                {
                    Surround::variable = 40;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class SecondWithin;

        static int
            variable;
};
```

Finally, we want to allow the nested classes to access each other's private members. Again this requires some friend declarations. In order to allow FirstWithin to access SecondWithin's private members nothing but a friend declaration in SecondWithin is required. However, to allow SecondWithin to access the private members of FirstWithin the friend class SecondWithin declaration cannot be plainly given in the class FirstWithin, as the definition of SecondWithin has not yet been given. A forward declaration of SecondWithin is required, and this forward declaration must be given in the class Surround, rather than in the class FirstWithin. Clearly, the forward declaration class SecondWithin in the class FirstWithin itself makes no sense, as this would refer to an external (global) class FirstWithin. But the attempt to provide the forward declaration of the nested class SecondWithin inside FirstWithin as class Surround::SecondWithin also fails miserably, with the compiler issuing a message like 'Surround' does not have a nested type named 'SecondWithin' The right procedure to follow here is to declare the class SecondWithin in the class Surround, before the class FirstWithin is defined. Using this procedure, the friend declaration of SecondWithin is accepted inside the definition of FirstWithin. The following class definition allows full access of the private members of all classes by all other classes:

```
class Surround
{
    class SecondWithin;
    public:
        class FirstWithin
        {
            friend class Surround;
            friend class SecondWithin;
            public:
                int getValue()
                {
        Surround::variable = SecondWithin::variable;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class FirstWithin;

        int getValue()
        {
      FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin
        {
            friend class Surround;
            friend class FirstWithin;
            public:
                int getValue()
                {
                                     Surround::variable =
FirstWithin::variable;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class SecondWithin;

        static int
            variable;
};
```

STUDENT ACTIVITY

1. What are different types of inheritance, describe with examples?

2. How can you create a virtual copy constructor?

NOTES

SUMMARY

- Member functions are as much a part of your class as the member variables. They determine what the objects of your class can do.
- The constructor is a class method with the same name as the class itself.
- Consequently, inline functions are always defined together with the class interface.
- Inline functions can be used when member functions consist of one very simple statement (such as the return statement in the function Person::getname()).
- The static functions can therefore address only the static data of a class; non-static data are unavailable to these functions.
- *Delete returns the memory to the free store. Remember that* the pointer itself—as opposed to the memory to which it points—is a local variable.
- Inheritance in programs made possible to reuse the attributes and functions of a parent class into a child class.
- The functions and attributes of a class are the combination of two things, its own functions and attributes and the functions r and attributes of all its super classes.
- Abstract classes are superclasses which contain *abstract methods* and are defined such that concrete subclasses are to extend them by implementing the *methods*.

SELF ASSESSMENT QUESTIONS

1. What is inheritance?

2. How do you show the declaration of a multiple class inheritance?

3. How do you invoke a base member function from a derived class in which you've overridden that function?

4. How do you invoke a base member function from a derived class in which you have not overridden that function?

5. If a base class declares a function to be virtual, and a derived class does not use the term virtual when overriding that class, is it still virtual when inherited by a third-generation class?

NOTES

UNIT 4 FUNCTION OVERLOADING AND POLYMORPHISM

★ LEARNING OBJECTIVES ★

- Polymorphism
- Function Overloading
- Operator Overloading
- Early Binding
- Polymorphism with Pointers
- Virtual Functions
- Late Binding and Pure Virtual Functions
- Opening and Closing of Files
- Stream Member Functions
- Binary File Operations
- Structures and File Operations
- Classes and File Operations
- Random Access File Processing

POLYMORPHISM

The word polymorphism has been derived from the greek word Polymorphous. Polus means (Many) and Morphous means (forms), so the meaning of polymorphism is many forms. You can relate polymorphism with synonyms of English language a single word can have multiple meanings. In C language you have already used function, but if you have multiple functions in a C program to perform same type of task you should think individual name for each functions. Using polymorphism C++ solved this problem, now a programmer can define same name functions in a program. Like you want to compute area of a rectangle, circle, triangle you can use area() name for each function. like:

- area(int length, int width)   // function for rectangle
- area(int radius)   // function for circle
- area(int base, int height)   // function for triangle

now you can see that in C++ same name functions can be used to perform different tasks.

In C++ two most popular forms of polymorphism are:

(*a*) Function Overloading

(*b*) Operator Overloading

FUNCTION OVERLOADING

In function overloading a same name functions can be used in a program to perform various tasks like:

- Print (int x)
- Print (char s)
- Print (float y)
- Print (emp e)

The Print() function is overloaded here.

- area(int length, int width)   // function for rectangle
- area(int radius)   // function for circle
- area(int base, int height)   // function for triangle

The area function is overloaded here. You can use area() function to compute area of different shapes like circle or triangle. On the other hand the meaning of operator overloading is use of single operator like (+ or >>) for different operations. Like:

```
X=a+b;                //add two integer values
Name3=sname+fname; // concat two strings
2.35          //use of . operator for.decimal place
3.67
emp.name= "mksharma" //use of.operator for object.
```

Program

```
// function overloading example
#include <iostream>
// Rectangle
double MomentOfInertia(double b, double h)
{
   return b * h * h * h / 3;
}
// Semi-Circle
double MomentOfInertia(double R)
{
   const double PI = 3.14159;
   return R * R * R * R * PI/ 8;
}
```

NOTES

```
    // Triangle
    double MomentOfInertia(double b, double h, int)
    {
        return b * h * h * h / 12;
    }
    void main()
    {
        double base = 7.74, height = 14.38, radius =
12.42;
        cout << "Rectangle\n"<< "Moment of inertia with
regard to the X axis: ";
        cout << "I = " << MomentOfInertia(base, height)
<< "mm\n\n";
        cout << "Semi-Circle\n" << "Moment of inertia
with regard to the X axis: ";
        cout << "I = " << MomentOfInertia(radius) <<
"mm\n\n";
        cout << "Enter the dimensions of the triangle\n";
        cout << "Base: ";   cin >> base;
        cout << "Height: "; cin >> height;
        cout << "\nTriangle\n" << "Moment of inertia with
regard to the X axis: ";
        cout << "I = " << MomentOfInertia(base, height,
1) << "mm\n\n";
}
```

Program

```
    //C++ program to overload function show(), to show
different values using the show() function
    #include <iostraem.h>
    void show(int val)
    {
        cout<<val;
    }
    void show(double val)
    {
        cout<<val;
    }
    void show(char *val)
```

```
{
    cout<< val;
}
    void main()
    {
    show(12);
    show(3.1415);
    show("Hello World\n!");
      }
```

OPERATOR OVERLOADING

Operator overloading allows C/C++ operators to have user-defined meaning in user defined class. Overloaded operators are part of C++ polymorphism. If you want to overload a defined operator like + or * to perform some user defined action then, the syntax is:

```
return type operator  + (value, values );
return type operator * (value , value);
example:
oload   operator + (oload, oload);    // overloading + operator
oload    operator * (oload, oload);    // overloading * operator
```

Program

```
class oload
  {
  public:
   // Without operator overloading:
    oload add(oload, oload);
    oload  mul(oload, oload);
   oload  f(oload a, oload b, oload c)
    {
      return add(add(mul(a,b), mul(b,c)), mul(c,a));
    }
    // With operator overloading:
    oload operator + (oload, oload);
    oload operator * (oload, oload);
   oload f(oload a, oload  b, oload  c)
    {
        return a*b + b*c + c*a;
    }
```

NOTES

Benefits of Operator Overloading

By overloading standard operators on a class, you can exploit the intuition of the users of that class. This lets users program in the language of the problem domain rather than in the language of the machine.

Examples of operator overloading

Few of examples of operator overloading:

- myString + yourString might concatenate two string objects
- myDate++ might increment a Date object
- a * b might multiply two Number objects
- a[i] might access an element of an Array object
- x = *p might dereference a "smart pointer" that actually "points" to a disk record

Overloaded operators

The following operators can be overloaded:

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	—	<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=	&=	|=
<<=	>>=	[]	()	->	->*	new	delete

However, some of these operators may only be overloaded as member functions within a class. This holds true for the '=', the '[]', the '()' and the '->' operators.

EARLY BINDING

When a C++ program is executed, it executes sequentially, beginning at the top of main(). When a function call is encountered, the point of execution jumps to the beginning of the function being called. How does the CPU know to do this?

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given it's own unique sequential address. This is no different for functions — when a function is encountered, it is converted into machine language and given the next available address. Thus, each function ends up with a unique machine language address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding

Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function.

Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember that all functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Let's take a look at a simple calculator program that uses early binding:

```
#include <iostream>
int Add(int nX, int nY)
{
    return nX + nY;
}
int Subtract(int nX, int nY)
{
    return nX - nY;
}

int Multiply(int nX, int nY)
{
    return nX * nY;
}

int main()
{
    int nX;
    cout << "Enter a number: ";
    cin >> nX;
    int nY;
    cout << "Enter another number: ";
```

```
  cin >> nY;

    int nOperation;
    do
    {
         cout << "Enter an operation (0=add, 1=subtract,
2=multiply): ";
        cin >> nOperation;
    } while (nOperation < 0 || nOperation > 2);

    int nResult = 0;
    switch (nOperation)
  {
        case 0: nResult = Add(nX, nY); break;
        case 1: nResult = Subtract(nX, nY); break;
        case 2: nResult = Multiply(nX, nY); break;
    }

    cout << "The answer is: " << nResult << endl;

    return 0;
}
```

Because Add(), Subtract(), and Multiply() are all direct function calls, the compiler will use early binding to resolve the Add(), Subtract(), and Multiply() function calls. The compiler will replace the Add() function call with an instruction that tells the CPU to jump to the address of the Add() function. The same holds true for Subtract() and Multiply().

POLYMORPHISM WITH POINTERS

In next program the main program defines pointers to the objects rather than defining the objects themselves in shown below:

```
vehicle *unicycle;
car     *sedan_car;
truck   *trailer;
boat    *sailboat;
```

Since we only defined pointers to the objects, we find it necessary to allocate the objects before using them by using the **new** operator in shown below:

```
unicycle = new vehicle;
...
sedan_car = new car;
...
trailer = new truck;
...
sailboat = new boat;
```

- Upon running the program, we find that even though we are using pointers to the objects, we have done nothing different than what we did in the first program.
- The program operates in exactly the same manner as the first program example. This should not be surprising because a pointer to a method can be used to operate on an object in the same manner as an object can be directly manipulated.
- Be sure to compile and run this program before continuing on to the next program example. In this program you will notice that we failed to check the allocation to see that it did allocate the objects properly, and we also failed to deallocate the objects prior to terminating the program.
- In such a simple program, it doesn't matter because the heap will be cleaned up automatically when we return to the operating system.
- In real program development you have to implement this allocation checking and the deallocation. As shown in the previous Module, if we do not deallocate, there will be garbage left.

Program

```
//Polymorphism with pointers
#include <iostream.h>
#include <stdlib.h>

//---base class declaration
//---and implementation part---
class vehicle
{
int wheels;
float weight;
public:
void message(void)
//first message()
```

NOTES

```
{cout<<"Vehicle message, from vehicle, the base class\n";}
};

//---derived class declaration and implementation part---
class car : public vehicle
{
int passenger_load;
public:
void message(void)  //second message()
{cout<<"Car message, from car, the vehicle derived class\n";}
};

class truck : public vehicle
{
int passenger_load;
float payload;
public:
int passengers(void) {return passenger_load;}
};

class boat : public vehicle
{
int passenger_load;
public:
int passengers(void) {return passenger_load;}
void message(void)  //third message()
{cout<<"Boat message, from boat, the vehicle derived class\n";}
};

//————————the main program————————
int main()
{
vehicle *unicycle;
car     *sedan_car;
truck   *trailer;
boat    *sailboat;
```

```

cout<<“Omitting the virtual keyword. Using\n”;
cout<<“pointer variables, and new keyword\n”;
cout<<“————————————————————————\n”;

unicycle = new vehicle;
unicycle->message();
sedan_car = new car;
sedan_car->message();
trailer = new truck;
trailer->message();
sailboat = new boat;
sailboat->message();

unicycle = sedan_car;
unicycle->message();



system(“pause”);
return 0;
}
```

VIRTUAL FUNCTIONS

A virtual function is a function that makes sure that, in an inheritance scenario, the right function is called regardless of the expression that calls the function. The late or dynamic binding is achieved in C++ with virtual functions. A function becomes virtual when its declaration starts with the keyword virtual. Once a function is declared virtual in a base class, its definition remains virtual in all derived classes; even when the keyword virtual is not repeated in the definition of the derived classes. Like:

```
virtual double Area() const;
virtual void show() const;
virtual void setweight(int wt);
```

Program

```
// C++ program to show the use of virtual function.
class Vehicle
```

NOTES

```
    {
        public:
            Vehicle();     // constructors
           Vehicle(int wt);  // interface.. now virtuals!
            virtual int getweight() const;
            virtual void setweight(int wt);
             private:
            int weight ;
    }
    // Vehicle's own getweight() function:
          int Vehicle::getweight() const
    {
        return (weight);
    }
    class Land: public Vehicle
    {
        ...
    }
    class Auto: public Land
    {
        ...
    }
    class Truck: public Auto
    {
        public:
            Truck();  // constructors
             Truck(int engine_wt, int sp, char const *nm,
int trailer_wt);
                            // interface: to set two weight
fields
           void setweight(int engine_wt, int trailer_wt);
                           // and to return combined weight
            int getweight() const;
        private:
            int trailer_weight;
    };
    // Truck's own getweight() function
    int Truck::getweight() const
    {
        return (Auto::getweight() + trailer_wt);
    }
```

Note that the keyword virtual appears only in the definition of the base class Vehicle; it need not be repeated in the derived classes. The effect of the late binding is:

```
    Vehicle
        v(1200);              // vehicle with weight 1200
    Truck  t(6000, 115,"Sawrajmazda", 25000);
    Vehicle  *vp;
    int main()
            {  //one
               vp = &v;
               cout<<vp->getweight();
               //two
               vp = &t;
               cout<<vp->getweight();
               cout<<vp->getspeed();
            }
```

Since the function getweight() is defined as virtual, late binding is used here: in the statements above, just below the one tag, Vehicle's function getweight() is called. In contrast, the statements below tag two use Truck's function getweight().

LATE BINDING AND PURE VIRTUAL FUNCTIONS

In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). In C++, one way to get late binding is to use function pointers. To review function pointers briefly, a function pointer is a type of pointer that points to a function instead of a variable. The function that a function pointer points to can be called by using the function call operator (()) on the pointer.

For example, the following code calls the Add() function:

```
int Add(int nX, int nY)
{
```

NOTES

```
    return nX + nY;
}

int main()
{
    // Create a function pointer and make it point to the
Add function
    int (*pFcn)(int, int) = Add;
    cout << pFcn(5, 3) << endl; // add 5 + 3

    return 0;
}
```

Calling a function via a function pointer is also known as an indirect function call. The following calculator program is functionally identical to the calculator example above, except it uses a function pointer instead of a direct function call:

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

int Add(int nX, int nY)
{
    return nX + nY;
}

int Subtract(int nX, int nY)
{
    return nX - nY;
}

int Multiply(int nX, int nY)
{
    return nX * nY;
}

int main()
{
    int nX;
    cout << "Enter a number: ";
    cin >> nX;

    int nY;
    cout << "Enter another number: ";
    cin >> nY;

    int nOperation;
    do
    {
        cout << "Enter an operation (0=add, 1=subtract,
2=multiply): ";
        cin >> nOperation;
    } while (nOperation < 0 || nOperation > 2);

     // Create a function pointer named pFcn (yes, the
syntax is ugly)
    int (*pFcn)(int, int);

    // Set pFcn to point to the function the user chose
    switch (nOperation)
    {
        case 0: pFcn = Add; break;
        case 1: pFcn = Subtract; break;
        case 2: pFcn = Multiply; break;
    }

    // Call the function that pFcn is pointing to with nX
and nY as parameters
    cout << "The answer is: " << pFcn(nX, nY) << endl;

    return 0;
}
```

In this example, instead of calling the Add(), Subtract(), or Multiply() function directly, we've instead set pFcn to point at the function we wish to call. Then we call the function through the pointer. The compiler is unable to use early binding to resolve the function call pFcn(nX, nY) because it cannot tell which function pFcn will be pointing to at compile time!

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the compiler can tell the CPU to jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

Implementing Pure Virtual Functions

Typically, the pure virtual functions in an abstract base class are never implemented. Because no objects of that type are ever created, there is no reason to provide implementations, and the ADT works purely as the definition of an interface to objects which derive from it. It is possible, however, to provide an implementation to a pure virtual function. The function can then be called by objects derived from the ADT, perhaps to provide common functionality to all the overridden functions.

In this example, the additional functionality is simply an additional message printed, but one can imagine that the base class provides a shared drawing mechanism, perhaps setting up a window that all derived classes will use.

```
//Implementing pure virtual functions

#include <iostream.h>

enum BOOL { FALSE, TRUE };

class Shape
{
public:
Shape(){}
~Shape(){}
virtual long GetArea() = 0; // error
virtual long GetPerim()= 0;
virtual void Draw() = 0;
private:
};

void Shape::Draw()
```

```
{
cout << "Abstract drawing mechanism!\n";
}

class Circle : public Shape
{
public:
Circle(int radius):itsRadius(radius){}
~Circle(){}
long GetArea() { return 3 * itsRadius * itsRadius; }
long GetPerim() { return 9 * itsRadius; }
void Draw();
private:
int itsRadius;
int itsCircumference;
};

void Circle::Draw()
{
cout << "Circle drawing routine here!\n";
Shape::Draw();
}


class Rectangle : public Shape
{
public:
Rectangle(int len, int width):
itsLength(len), itsWidth(width){}
~Rectangle(){}
long GetArea() { return itsLength * itsWidth; }
long GetPerim() {return 2*itsLength + 2*itsWidth; }
virtual int GetLength() { return itsLength; }
virtual int GetWidth() { return itsWidth; }
void Draw();
private:
```

```
int itsWidth;
int itsLength;
};

void Rectangle::Draw()
{
for (int i = 0; i<itsLength; i++)
{
for (int j = 0; j<itsWidth; j++)
cout << "x ";

cout << "\n";
}
Shape::Draw();
}


class Square : public Rectangle
{
public:
Square(int len);
Square(int len, int width);
~Square(){}
long GetPerim() {return 4 * GetLength();}
};

Square::Square(int len):
Rectangle(len,len)
{}

Square::Square(int len, int width):
Rectangle(len,width)

{
if (GetLength() != GetWidth())
cout << "Error, not a square... a Rectangle??\n";
```

```
}

int main()
{
int choice;
BOOL fQuit = FALSE;
Shape * sp;

while (1)
{
cout << “(1)Circle (2)Rectangle (3)Square (0)Quit:”;
cin >> choice;

switch (choice)
{
case 1: sp = new Circle(5);
break;
case 2: sp = new Rectangle(4,6);
break;
case 3: sp = new Square (5);
break;
default: fQuit = TRUE;
break;
}
if (fQuit)
break;

sp->Draw();
cout << “\n”;
}
return 0;
}
```

OPENING AND CLOSING OF FILES

File Streams

File Stream provide a uniform way of dealing with data coming from the hard disk and going out to the screen or printer or coming from the keyboard and going to hard disk. In either case, you can use the insertion and extraction operators with file stream objects with related functions. To open and close files, you have ofstraem, ifstream and fstream objects.

• Ofstraem	: to write into files
• Ifstream	: to read from files
• Fstream	: both read and write

Ofstream and Ifstream

The ofstream used to read from or write to files are called ofstream objects. These are derived from the iostream objects you've been using so far. To get started with writing to a file, you must first create an ofstream object, and then associate that object with a particular file on your disk. To use ofstream objects, you must be sure to include fstream.h in your program.

While the ifstream is used read data values from a file. To get started to read from a file, you must first create an ifstream object, and then associate that object with a particular file on your disk. To open the file myfile.txt with an ofstream object, declare an instance of an ofstream object and pass in the filename as a parameter:

ofstream fout("myfile.txt");

Opening this file for input works exactly the same way, except it uses an ifstream object:

ifstream fin("myfile.txt");

Program

```
     #include <fstream.h>
     void  main()
     {
        char fileName[80];
        char buffer[255];      // for user input
        cout << "Enter File name: ";
       cin >> fileName;
        ofstream fout(fileName);  // open for writing
          fout << "This line written directly to the
file...\n";
       cout << "Enter text for the file: ";
        cin.ignore(1,'\n');  // ignore the newline after
the file name
        cin.getline(buffer,255);  // get the user's input
        fout << buffer << "\n";    // and write it to the
file
       fout.close();                 // close the file,
ready for reopen
       ifstream fin(fileName);     // reopen for reading
        cout << "Here's the contents of the file:\n";
        char ch;
        while (fin.get(ch))
        cout << ch;
        cout << "\n***End of file ***\n";
        fin.close();              // close the file stream
 }
```

File Opening Modes

The default behaviour upon opening a file is to create the file if it doesn't yet exist and to truncate the file or delete all its contents if it does exist. If you don't want this default behaviour, you can explicitly provide a second argument to the constructor of your ofstream object.Valid arguments include:

- **ios::app**—Appends to the end of existing files rather than truncating them.
- **ios::at**—Places you at the end of the file, but you can write data anywhere in the file.
- **ios::trun**—The default. Causes existing files to be truncated.
- **ios::nocreat**—If the file does not exist, the open fails.
- **ios::noreplac**—If the file does already exist, the open fails.

Note that app is short for append; ate is short for at end, and trunc is short for truncate.

Program

```
                          // program to show the  Appending
of data at  the end of a file
        #include <fstream.h>
       void  main()
     {
        char fileName[80];
        char buffer[255];
       cout << "Please re-enter the file name:";
       cin >> fileName;
        ifstream fin(fileName);
```

NOTES

```
    if (fin)                    // already exists?
    {
        cout << "Current file contents:\n";
       char ch;
       while (fin.get(ch))
           cout << ch;
        cout << "\n***End of file contents.***\n";
     }
      fin.close();
      cout << "\nOpening" << fileName << " in append
mode...\n";
     ofstream fout(fileName,ios::app);
     if (!fout)
     {
        cout << "Unable to open " << fileName << " for
appending.\n";
        return(1);
     }
     cout << "\nEnter text for the file:";
     cin.ignore(1,'\n');
     cin.getline(buffer,255);
      fout << buffer << "\n";
     fout.close();
   fin.open(fileName);  // reassign existing fin object!
     if (!fin)
     {
         cout << "Unable to open" << fileName << " for
reading.\n";
       return(1);
    }
     cout << "\nHere's the contents of the file:\n";
     char ch;
     while (fin.get(ch))
     cout << ch;
    cout << "\n***End of file contents.***\n";
     fin.close();
        }
```

STREAM MEMBER FUNCTIONS

Every C++ program that includes the iostream classes has four objects that are created and initialized. When iostream class library is added to your program you can use all the functions to put the appropriate include statement at the top of your program listing. Like:

cin: handles input from the standard input, the keyboard.

cout: handles output to the standard output, the screen.

Cer: handles un buffered output to the standard error device, the screen. Because this is un buffered, everything sent to cerr is written to the standard error device immediately, without waiting for the buffer to fill or for a flush command to be received.

clog: handles buffered error messages that are output to the standard error device, the screen. It is common for this to be "redirected" to a log file, as described in the following section.

Read Data Values

The object cin is responsible for read or input data values and is made available to your program when you include iostream.h. Using the overloaded extraction operator (>>) cin can put data into your program's variables. Like:

int someVariable;

cout << "Enter a number:";

cin >> someVariable;

You should learn now that cin can overloaded the extraction operator for a great variety of parameters, among them int&, short&, long&, double&, float&, char&, char*, and so forth. When you write:

cin >> someVariable;

the type of someVariable is assessed. In the example above, someVariable is an integer, so the following function is called:

istream & operator>> (int &)

Note that because the parameter is passed by reference, the extraction operator is able to act on any type of C++ original variable like:

Program

```
#include <iostream.h>
void main()
 {
   int myInt;
   long myLong;
  double myDouble;
```

NOTES

```
      float myFloat;
      unsigned int myUnsigned;
     cout << "int:";
     cin >> myInt;
     cout << "Long:";
     cin >> myLong;
    cout << "Double:";
    cin >> myDouble;
    cout << "Float:";
    cin >> myFloat;
    cout << "Unsigned:";
   cin >> myUnsigned;
    cout << "\n\nInt:\t" << myInt << endl;
    cout << "Long:\t" << myLong << endl;
    cout << "Double:\t" << myDouble << endl;
   cout << "Float:\t" << myFloat << endl;
    cout << "Unsigned:\t" << myUnsigned << endl;
}
```

String Handling Problem

Using cin, when you will try to enter a full name into a string. cir believes that white space is a separator. When it sees a space or a new line, it assumes the input for the parameter is complete, and ir the case of strings it adds a null character right then and there anc you cannot input two strings separated using simple cin like "ml sharma". In above example you can check it.

Example

```
                    // string problem using cin
       #include <iostream.h>
       void main()
             {
           char YourName[50];
           cout << "Your first name: ";
          cin >> YourName;
          cout << "Here it is: " << YourName << endl;
          cout << "Your Full name:";
         cin >> YourName;
        cout << "Here it is: " << YourName << endl;
        }
```

```
Output: Your first name: Mahesh
Here it is: Mahesh
Your Full name: Mahesh Kumar Sharma
Here it is: Mahesh Kumar Sharma
```

Get() with Cin

The cin Operator >> taking a character reference can be used to get a single character, multiple characters or strings from the standard input. That you will check in given examples:

Example

```
#include <iostream.h>
 void main()
 {
   char ch;
    while ( (ch = cin.get()) != EOF)
    {
       cout << "ch: " << ch << endl;
   }
   cout << "\nDone!\n";
}
```

to exit this program, you must send end of file from the keyboard. On DOS /windows computers use Ctrl+Z .

```
World
ch: W
ch: o
ch: r
ch: l
ch: d
ch:
(ctrl-z)
Done!
```

Example

```
                    // Read  multiple characters with cin
 void  main()
     {
       char a, b, c;
       cout << "Enter three letters:";
```

NOTES

```
        cin.get(a).get(b).get(c);
      cout << "a: " << a << "\nb: " << b << "\nc: " << c <<
endl;
      }
```

```
Output: Enter three letters: mks
a: m
b: k
c: s
```

Example

```
                //Read strings with cin
void  main()
     {
            char stringOne[256];
            char stringTwo[256];
           cout << "Enter string one: ";
           cin.get(stringOne,256);
          cout << "stringOne: " << stringOne << endl;
         cout << "Enter string two: ";
        cin >> stringTwo;
       cout << "StringTwo:" << stringTwo << endl;
 }
```

```
Output: Enter string one: My name is mks
stringOne: My name is mks
Enter string two: What is yours
StringTwo: What
```

getline(), putline()

When a user want to enter a string, and that string can be read by getline(). Like get(), getline() takes a buffer and a maximum number of characters. Unlike get(), however, the terminating newline is read and thrown away. With get() the terminating newline is not thrown away. It is left in the input buffer. You can use putline() to print the string on screen with spaces.

Example

```
     #include <iostream.h>
     void  main()
     {
        char sOne[256];
        char sTwo[256];
       char sThree[256];
       cout << "Enter string one:";
      cin.getline(sOne,256);
     cout << "stringOne:" << sOne << endl;
   cout << "Enter string two: ";
    cin >> sTwo;
    cout << "stringTwo:" << sTwo << endl;
    cout << "Enter string three:";
    cin.getline(sThree,256);
    cout << "stringThree:" << sThree << endl;
  }

  Output: Enter string one: one two three
  stringOne: one two three
  Enter string two: four five six
  stringTwo: four
  Enter string three: stringThree: five six
```

gnore(), peek() and putback()

\ny times you want to ignore the remaining characters on a line ıntil you hit either *end of line* (EOL) *or end of file* (EOF). The member function ignore() serves this purpose. Ignore() takes two parameters, the maximum number of characters to ignore and the termination character. If you write ignore(80,'\n'), up to 80 characters will be thrown away until a newline character is found. The input object cin has two additional methods that can be used in some programs peek(), which looks at but does not extract the next character, and putback(), which inserts a character into the input stream.

Program

```
#include <iostream.h>
void main()
{
     char ch;
    cout << "enter a phrase:";
    while ( cin.get(ch) )
    {
```

NOTES

```
            if (ch == '!')
               cin.putback('$');
           else
               cout << ch;
             while (cin.peek() == '#')
                cin.ignore(1,'#');
         }
       }
       Output: enter a phrase: Now!is#the!time#for!fun#!
       Now$isthe$timefor$fun$
```

put(), write()

You have used cout along with the overloaded insertion operator (<<) to write strings, integers, and other numeric data to the screen. It is also possible to format the data, aligning columns and writing the numeric data in decimal and hexadecimal. Just as the extraction operator can be supplemented with get() and getline(), the insertion operator can be supplemented with put() and write(). *The function put() is* used to write a single character to the output device. Because put() returns an ostream reference, and because cout is an ostream object, you can concatenate put() just as you do the insertion operator.

Example

```
// use of put with cout
#include <iostream.h>
 void  main()
      {
      cout.put('H').put('e').put('l').put('l').put('o').put('\n');
}
output: Hello
```

Example

```
        //  program to show the use of write() function
with cout
       #include <iostream.h>
       #include <string.h>
       void main()
      {
        char One[] = "India is my land";
        int fullLength = strlen(One);
        int tooShort = fullLength -7;
       int tooLong = fullLength + 5;
       cout.write(One,fullLength) << "\n";
       cout.write(One,tooShort) << "\n";
       cout.write(One,tooLong) << "\n";
 }
Output: India is my motherland
India is
India is my motherland i?!.!
```

NOTES

width(), fill()

The default width of your output will be just enough space to print the number, character, or string in the output buffer. You can change this by using width(). Normally cout fills the empty field created by a call to width() with spaces. At times you may want to fill the area with other characters, such as asterisks or +. To do this, you call fill() and pass in as a parameter the character you want used as a fill character. Like:

Example

```
    #include <iostream.h>
    void  main()
    {
      cout << "aimca >";
      cout.width(25);
      cout << courses << "< MCA\n";
      cout << "aimca >";
      cout.width(25);
      cout.fill('*');
      cout << courses << "< MBA\n";
 }
Output: aimca >                    courses< MCA
aimca >*****************courses<  MBA
```

setf()

NOTES

The iostream objects keep track of their state by using flags. You can set these flags by calling setf() and passing in one or another of the predefined enumerated constants. For example, you can set whether or not to show trailing zeroes (so that 20.00 does not become truncated to 20). To turn trailing zeroes on, use setf(ios::showpoint).You can turn on the plus sign (+) before positive numbers by using ios::showpos. You can change the alignment of the output by using ios::left, ios::right, or ios::internal.Finally, you can set the base of the numbers for display by using ios::dec (decimal), ios::oct (octal—base eight), or ios::hex (hexadecimal—base sixteen). Like:

Example

```
#include <iostream.h>
#include <iomanip.h>
void  main()
{
  const int number = 185;
  cout << "The number is" << number << endl;
  cout << "The number is" << hex <<  number << endl;
   cout.setf(ios::showbase);
  cout << "The number is" << hex <<  number << endl;
  cout << "The number is";
  cout.width(10);
  cout << hex << number << endl;
  cout << "The number is";
  cout.width(10);
  cout.setf(ios::left);
  cout << hex << number << endl;
  cout << "The number is";
  cout.width(10);
  cout.setf(ios::internal);
  cout << hex << number << endl;
   cout << "The number is:" << setw(10) << hex <<
number << endl;
 }
```

BINARY FILE OPERATIONS

Operating systems, such as DOS or Windows, distinguish between text files and binary files. Text files store everything as text large numbers such as 54, 325 are stored as a string of numerals ('5', '4', ',', '3', '2', '5'). This can be inefficient, but has the advantage that the text can be read using simple programs such as the DOS command type or Unix command cat or in Windows using notepad. Today there is a need to store images, sounds, video in a file form. To help this file system distinguish between text and binary files, C++ provides the ios::binary flag to create binary files. On many systems, this flag is ignored because all data is stored in binary format. Binary files can store not only integers and strings, but entire data structures or class can be write or read at once in a binary file using write() and read() methods. Like in a class employee to write or read an object you can use:

fout.write(char* &name of object ,sizeof (object));

fout.write(char* &emp,sizeof (emp));

fout.read(char* &name of object ,sizeof (object));

fout.read(char* &emp,sizeof (emp));

Each of these functions expects a pointer to character, however, so you must cast the address of your class to be a pointer to character. The second argument to these functions is the number of characters to write, which you can determine using sizeof() function.

Example

```
// program to write and read  data of an employee in text
file
#include <fstream.h>
#include <iostream.h>
#include <string.h>
void  main()
{
char FileName[20];
char EName[40], Address[50], City[20], State[32],
pinCode[10];
   cout << "Enter the Following pieces of information\n";
    cout << "Employee Name:";
    cin.getline(EName, 40);
    cout << "Address:  ";
    cin.getline(Address, 50);
    cout << "City:      ";
    cin.getline(City, 20);
    cout << "State:     ";
    cin.getline(State, 32);
```

```
     cout << "Pin Code:";
     cin.getline(pinCode, 10);
      cout << "\nEnter the name of the file you want to
create:";
     cin >> FileName;
     ofstream EmplRecords(FileName, ios::out);
   EmplRecords << EName << "\n" << Address << "\n" << City
   << "\n" << State << "\n" << ZIPCode;
      cout << "Enter the name of the file you want to
open:";
     cin >> FileName;
     ifstream EmplRecords(FileName);
     EmplRecords.getline(EName, 40, '\n');
     EmplRecords.getline(Address, 50);
     EmplRecords.getline(City, 20);
     EmplRecords.getline(State, 32);
     EmplRecords.getline(ZIPCode, 10);
     cout << "\n -=- Employee Information -=-";
     cout << "\nEmpl Name: " << EName;
     cout << "\nAddress:   " << Address;
     cout << "\nCity:      " << City;
     cout << "\nState:     " << State;
     cout << "\nZIP Code:  " << ZIPCode;
     cout << "\n\n";
}
```

Example

```
//program to write a block of class object and read with
the help of write() and read function
     #include <fstream.h>
       class Animal
      {
public:     //constructor defined
Animal(int weight, long days):itsWeight(weight),
itsNumberDaysAlive(days){}
          ~Animal(){}  // use of destructor
          int GetWeight()const { return itsWeight; }
       void SetWeight(int weight) { itsWeight = weight;}
   long GetDaysAlive()const { return  itsNumberDaysAlive;}
       void SetDaysAlive(long days) { itsNumberDaysAlive
= days; }
    private:
       int itsWeight;
       long itsNumberDaysAlive;
    };
   void  main()
    {
       char fileName[80];
       char buffer[255];
       cout << "Please enter the file name:";
       cin >> fileName;
       ofstream fout(fileName,ios::binary);
       if (!fout)
       {
          cout << "Unable to open" << fileName << "for
writing.\n";
          return(1);
      }
      Animal Dog(50,30);
      fout.write((char*) &Dog,sizeof Dog);
      fout.close();
      ifstream fin(fileName,ios::binary);
      if (!fin)
      {
         cout << "Unable to open " << fileName << " for
reading.\n";
         return(1);
      }
      Animal DogTwo(1,1);
         cout << "DogTwo weight:" << DogTwo.GetWeight()
<< endl;
         cout << "DogTwo days:" << DogTwo.GetDaysAlive()
<< endl;
       fin.read((char*) &DogTwo, sizeof DogTwo);
```

NOTES

NOTES

```
        cout << "DogTwo weight:" << DogTwo.GetWeight()
<< endl;
        cout << "DogTwo days:" << DogTwo.GetDaysAlive()
<< endl;
        fin.close();
 }
```

STRUCTURES AND FILE OPERATIONS

C++ File I/O with binary files using fstream class is a simple task. fstream class has the capability to do both Input as well as Output operations *i.e.,* read and write. All types of operations like reading/ writing of characters, strings, lines and not to mention buffered I/O are supported by fstream. Operating systems store the files in binary file format. Computers can deal with only binary numbers. But binary files are not readable by humans. Our level of comfort lies only with proper ASCII or UNICODE characters. This article deals with how C++ File I/O class fstream can be used for reading and writing binary files. *For ASCII file operations in C++, refer to* C++ Text file I/O article. For our C++ File I/O binary file examples, now assume a **struct WebSites** with two members as follows.

```
      // Struct for C++ File I/O binary file sample
      struct WebSites
      {
           char SiteName[100];
           int Rank;
      };
```

Write operations in C++ Binary File I/O

There are some important points to be noted while doing a write operation.

- The file has to be opened in output and binary mode using the flags ios::out (output mode) and ios::binary(binary mode)
- The function *write* takes two parameters. The first parameter is of type *char* * for the data to be written and the second is of type *int* asking for the size of data to be written to the binary file.
- File has to be closed at the end.

```
      // Sample for C++ File I/O binary file write
    void write_to_binary_file(WebSites p_Data)
```

```
                fstream  binary_file("c:\\test.dat",
ios::out|ios::binary|ios::app);
            binary_file.write(reinterpret_cast<char
*>(&p_Data),sizeof(WebSites));
            binary_file.close();
    }
```

The above C++ File I/O binary sample function writes some data to the function. The file is opened in output and binary mode with *ios::out* and *ios::binary*. There is one more specifier *ios::app*, which tells the Operating system that the file is also opened in *append* mode. This means any new set of data will be appended to the end of file. Also the *write* function used above, needs the parameter as a character pointer type. So we use a type converter reinterpret_cast to typecast the structure into char* type.

Read Operations in C++ Binary File I/O

This also has a similar flow of operations as above. The only difference is to open the file using ios::in, which opens the file in read mode.

```
        // Sample for C++ File I/O binary file read
         void read_from_binary_file()
         {
             WebSites p_Data;
                fstream  binary_file("c:\\test.dat",
ios::binary|ios::in);
               binary_file.read(reinterpret_cast<char
*>(&p_Data),sizeof(WebSites));
             binary_file.close();
             cout<<p_Data.SiteName<<endl;
             cout<<"Rank  :"<< p_Data.Rank<<endl;


         }
```

CLASSES AND FILE OPERATIONS

Writing a class to a file

1. #include <fstream.h>
2.
3. class Animal

```
{
public:
Animal(int weight, long days):itsWeight(weight), itsNumberDaysAlive(days){}
~Animal(){}

int GetWeight()const { return itsWeight; }
void SetWeight(int weight) { itsWeight = weight; }

long GetDaysAlive()const { return itsNumberDaysAlive; }
void SetDaysAlive(long days) { itsNumberDaysAlive = days; }

private:
int itsWeight;
long itsNumberDaysAlive;
};

int main()   // returns 1 on error
{
char fileName[80];
char buffer[255];

cout << “Please enter the file name:”;
cin >> fileName;
ofstream fout(fileName,ios::binary);
if (!fout)
{
cout << “Unable to open” << fileName << “ for writing.\n”;
return(1);
}

Animal Bear(50,100);
fout.write((char*) &Bear,sizeof Bear);

fout.close();

ifstream fin(fileName,ios::binary);
if (!fin)
{
cout << "Unable to open" << fileName << " for reading.\n";
return(1);
}

Animal BearTwo(1,1);

cout << "BearTwo weight:" << BearTwo.GetWeight() << endl;
cout << "BearTwo days:" << BearTwo.GetDaysAlive() << endl;

fin.read((char*) &BearTwo, sizeof BearTwo);

fin.close();
return 0;
}
```

RANDOM ACCESS FILE PROCESSING

A binary file is a file of any length that holds *bytes* with values in the range 0 to 0xff. (0 to 255). These bytes have no other meaning. In a text file a value of 13 means carriage return, 10 means line feed, 26 means end of file. Software reading or writing text files has to deal with line ends. In *Linux* these are just separated by line feeds but Windows uses carriage returns and line feeds.

In modern terms we can call a binary file a stream of bytes and more modern languages tend to work with streams rather than files. The important part is the data rather than where it came from! This example shows that you can write text to a binary file.

```
RandomAccess ra( filename) ;


if ( ra.OpenWrite() )


  {
    if (!ra.Write( mytext ))
cout << "Failed to write to file " << filename << endl;

    ra.Close() ;
  }
else
    cout << "Failed to open" << filename << " fior writing"
<< endl;
```

NOTES

This uses the class RandomAccess to open a binary file for writing, then writes a string into it. The RandomAccess class uses a FILE to do the main work. It's opened in "wb " mode (refer to the C tutorial for more information on that) and then writes the text to the file. It's actually writing sequentially though it could be made to write anywhere in the file.

STUDENT ACTIVITY

1. What are the common operators for overload?

2. What is a ofstream object?

NOTES

SUMMARY

- The word polymorphism has been derived from the greek word Polymorphous.
- Operator overloading allows C/C++ operators to have user-defined meaning in user defined class.
- **Early binding** (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address.
- A virtual function is a function that makes sure that, in an *inheritance scenario, the right function is called regardless of* the expression that calls the function.
- The ofstream used to read from or write to files are called ofstream objects.
- Binary files can store not only integers and strings, but entire data structures or class can be write or read at once in a binary file using write() and read() methods.

SELF ASSESSMENT QUESTIONS

1. When you overload member functions, in what ways must they differ?
2. What is the difference between function and operator overloading, describe with the help of example?
3. When is the destructor called?
4. How *does the copy constructor differ from the assignment operator* (=)?
5. What is the this pointer?
6. How do you differentiate between overloading the prefix and postfix increment operators?
7. Can you overload the operator+ for short integers?
8. Is it legal in C++ to overload the operator++ so that it decrements a value in your class?
9. What return value must conversion operators have in their declarations?
10. What is a stream, how can you differ input stream and output stream?
11. What is fstream, and what does it do?
12. What are the three forms of cin.get(), and what are their differences?
13. What *is the difference between cin.read() and cin.getline()?*
14. What are the different file opening modes?
15. Write about command line parameters.
16. What does the ios::app argument do?