Chapter 1

Overview of Programming Approaches

1.1 Overview of Procedure-Oriented Programming (POP)/ Structured Programming

Conventional Programming, using high level languages such as COBOL (Common Business Oriented Language), FORTAN (Formula Translation) and C, is commonly called as procedural oriented programming (POP).

It enforces a top-down design model, in which developers map out the overall program structure into separate subsections to make programs more efficient and easier to understand and modify. In this technique, program flow follows a simple hierarchical model that employs three types of control flows: sequential, selection, and iteration.

POP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as function. A typical program structure for procedural programming is shown in the figure below.

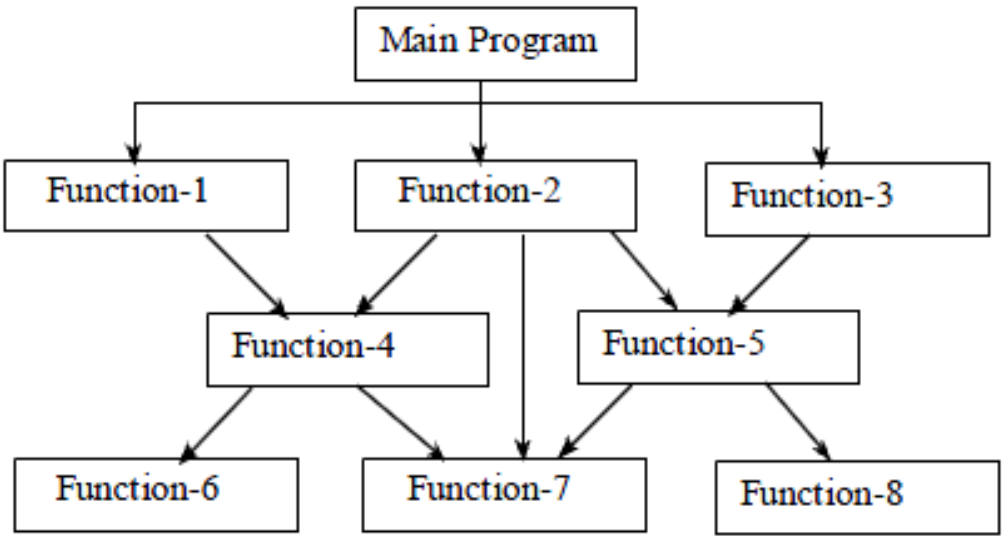


Fig: Typical Structure of Procedure Oriented Programs

In a multi-function program, many important data items are placed as globally. So that they may be accessed by all the functions. Each function may have its own local data. The figure shown below shows the relationship of data and function in a procedure-oriented program.

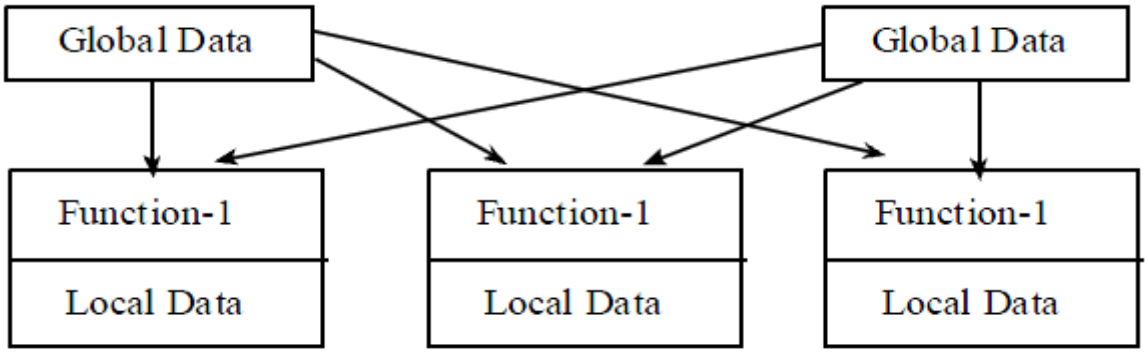


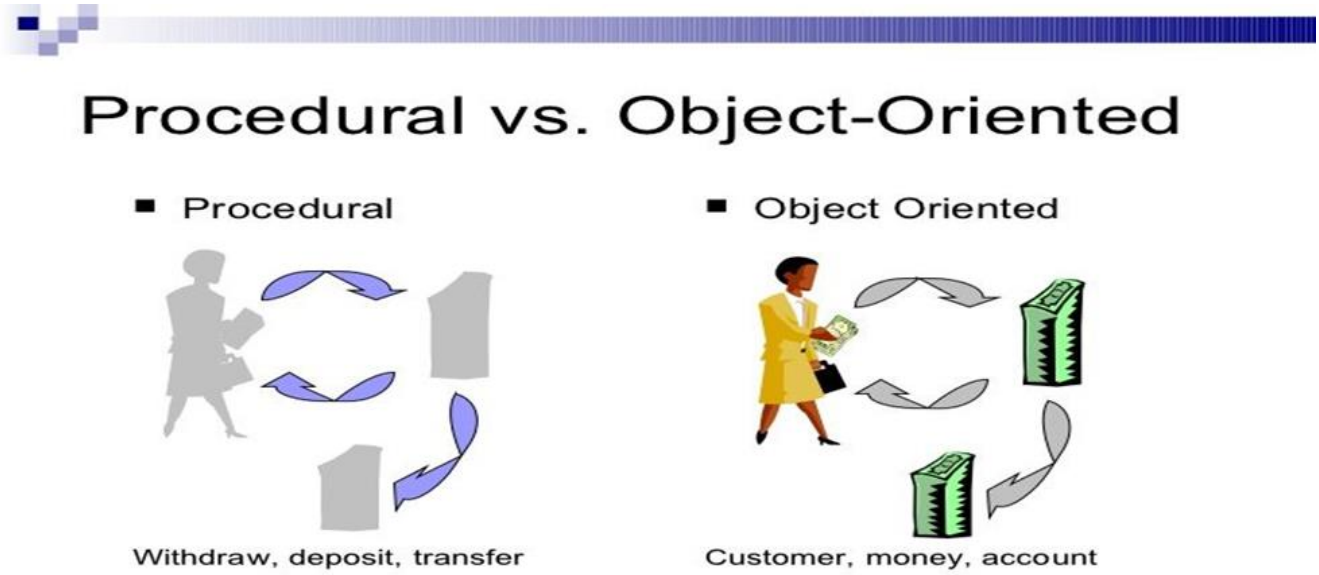
Fig: Relationship of data and functions in procedure programming

Some features of procedure-oriented programming are:

- Emphasis is on doing things (algorithms), procedures.
- Large programs are divided into smaller programs called as functions.
- Most of the functions share global data.
- Data and Functions don't tide with each other.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Some issues with procedural programming are:

- **Data Undervalued:** Data is given second-class status in the organization of procedural languages.
- **Insecure Data**: A global data can be corrupted by functions. Since many functions access the same global data, the way the data is stored becomes critical.
- **Relationship to the Real World:** Procedural programs are often difficult to design because their chief components – functions and data structures – don't model the real world very well.
- **New Data Types:** It is difficult to create new data types with procedural languages.
- **Complex and time consuming:** Does not provide code reusability, hence requires lots of coding work.
- **Does not support generic programming:** One can't write programs independent of particular data types.



1.2 Object-oriented Programming

- The object-oriented programming is an approach that combines or encapsulates both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object.
- The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data.
- In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data.
- An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called message passing.
- Examples of object-oriented languages include C++, Python, C#, Java etc.

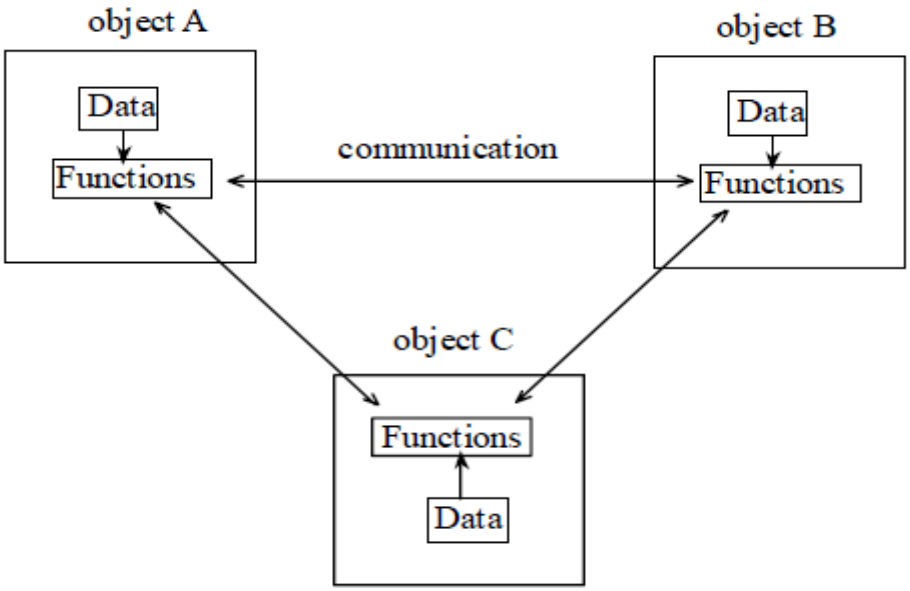


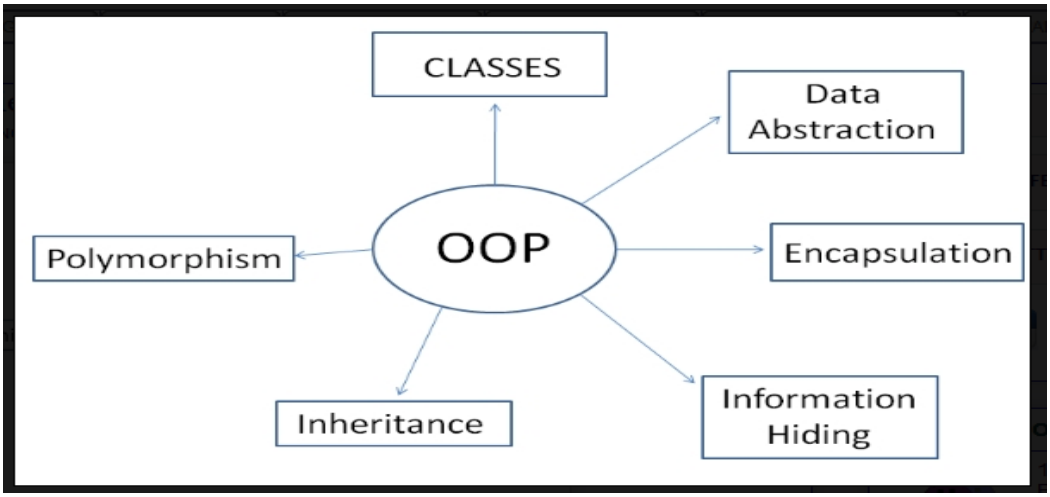
Fig: Organization of data & functions in OOP

In object oriented programming:

- Emphasis is on data rather than procedure.
- Programs are divided into 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 functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

1.2.1 Characteristics/features of Object-Oriented Language:

- Objects
- Classes
- Data Abstraction
- Encapsulation, and Data Hiding
- Inheritance
- Polymorphism
- Dynamic Binding



Objects: This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object. Objects are the basic run-time entities in an object-oriented language. Objects are instances of classes. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory.

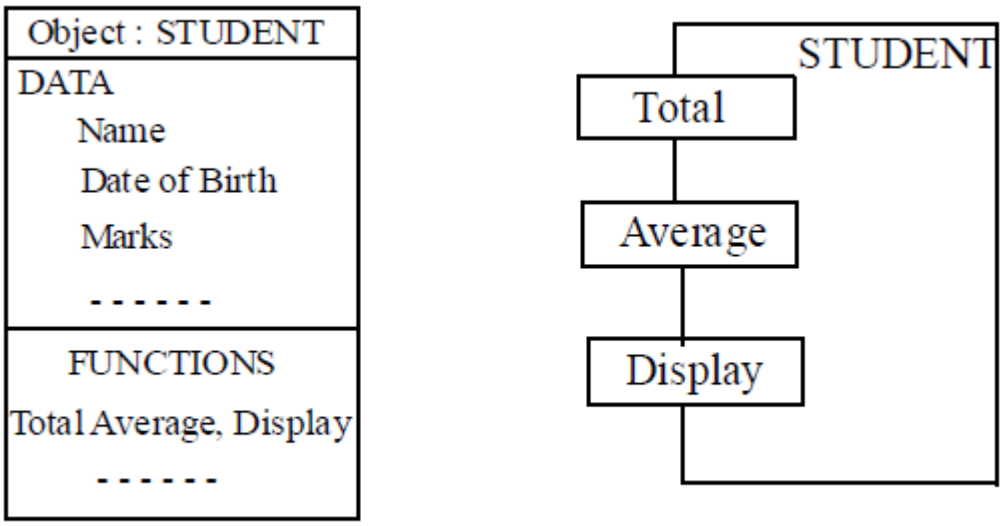


Fig: Two way of representing an object

Classes: Class is a user-defined data type. It is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. E.g.: grapes, bananas and orange are the member of class fruit. Classes are user-defined data types and behave like the built-in types of a programming language.

Data Encapsulation and Data Abstraction: Combining data and functions into a single unit (class) is known as Encapsulation. ***Data encapsulation*** is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those functions which are present in the class can access the data. The insulation of the data from direct access by the program is called ***data hiding*** or information hiding.

Hiding the complexity of program (*size, cost*) is called ***Abstraction*** and only essential features are represented. In short we can say that internal working is hidden. In C++, classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values.

1. **Abstraction using Classes**: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.
2. **Abstraction in Header files**: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.
3. **Abstraction using access specifiers:** Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example: Members declared as public in a class, can be accessed from anywhere in the program. Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

   We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Inheritance: It is the process by which object of one class acquire the properties or features of objects of another class. The concept of inheritance provides the idea of reusability, means we can add additional features to an existing class without modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class (called derived class) from another class (called base class), some of the data and functions can be inherited so that we can reuse the already written and tested code in our program, simplifying our program. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles.

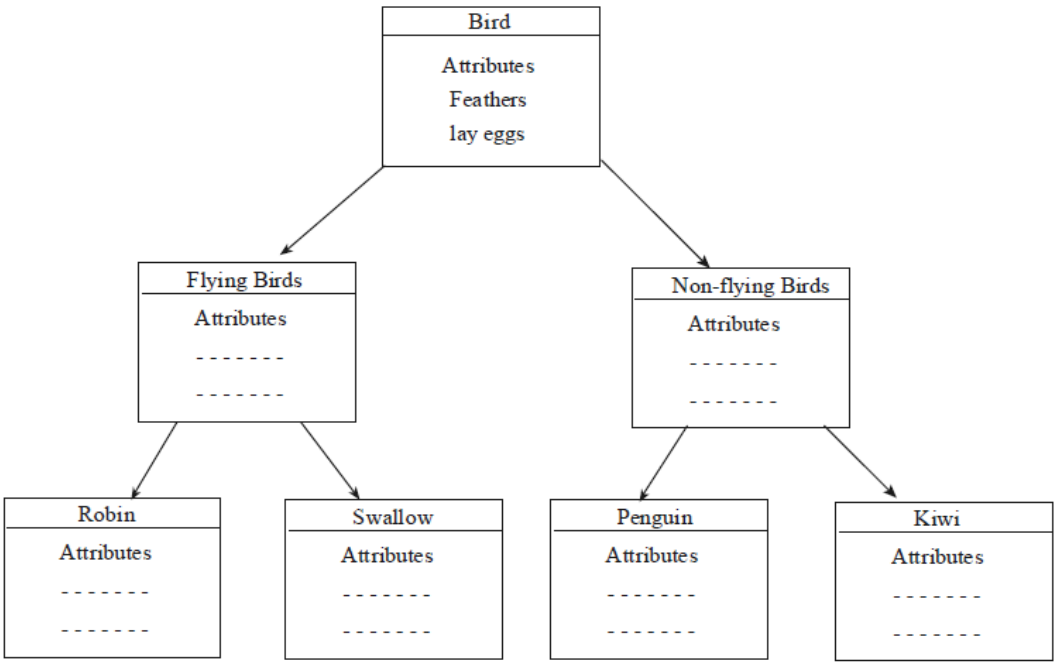


Fig: Property Inheritance

Polymorphism: Polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

Example:

- *Operator Overloading:* An operator represents different behaviors in different instances is known as operator overloading. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. Existing set of operators can be used to operate with user defined types. One can provide additional meaning to the existing set of operators.
- *Function Overloading:* Using a single function name we can deal with different type/number of arguments, it is called function overloading. A single function name can be used to handle different number and different types of arguments as in figure.

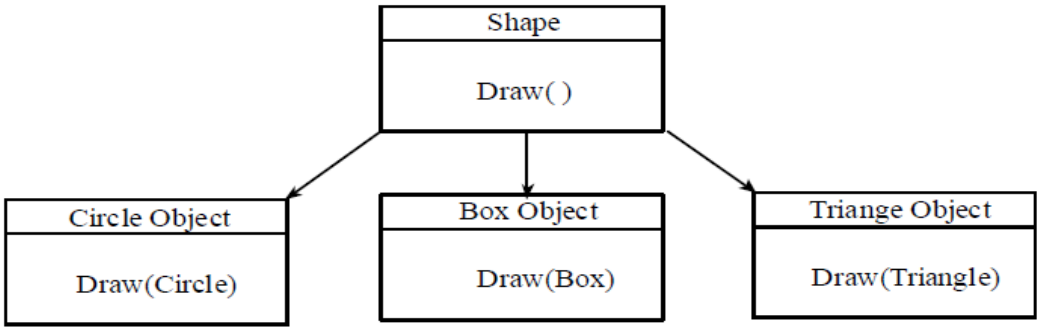


Fig: Polymorphism

Dynamic Binding: Binding refers to linking of function call with function definition. And when binding takes place at run-time, it is called dynamic binding. OOP allows dynamic binding. The code associated to the function call is not known during compile time and it is only known during the execution. Dynamic binding is associated with polymorphism and inheritance.

1.2.2 Advantages of Object-oriented Programming

- ❖ Programs are easy to develop.
- ❖ Data hiding is achieved through encapsulation.
- ❖ Data centered approach rather than process centered approach.
- ❖ Elimination of redundant code due to inheritance, that is, we can re-use the code by deriving a new class from existing one.

1.3 C versus C++

C	C++
C Follows the procedural programming approach, importance is given to the procedure of the program.	C++ is an OOP language, importance is given to the data rather than process.
Does not provide data hiding facility .i.e., data is secure.	Provides data hiding facility through encapsulation i.e., data is secure.
Functions are the building blocks of a C program.	Objects are building blocks of a C++ program.
C doesn't support function overloading.	C++ enables function overloading.
Structures cannot contain functions in C.	In C++, functions can be used inside a structure.
Provides little code reusability, thus hard to write complex programs.	Extreme code reusability through inheritance, so easier to write complex programs.
Complex dynamic memory allocation (malloc(), calloc(), free()).	Efficient and easy dynamic memory allocation through new and delete operators.
C uses top-down approach.	C++ follows bottom-up approach.

Chapter 2

C++ Basics

2.1 A Simple C++ Program

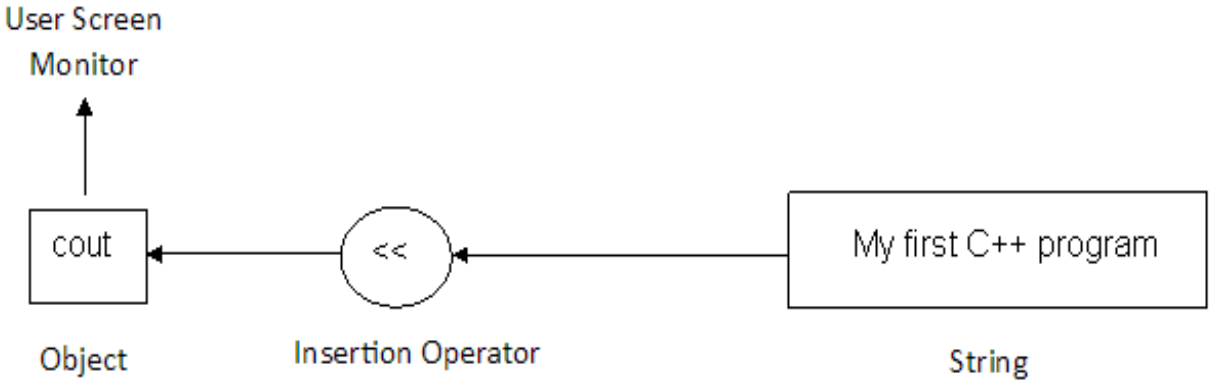
```
#include <iostream>          //allows program to accept data from keyboard, and output data to the screen
using namespace std;

int main()                              // main() is where program execution begins.

{
        cout << " My first C++ program ";   // displays the content on the screen
        return 0; //indicates that the program is terminated successfully
}
```

This program demonstrates following C++ features:

1. **#include <iostream>**: *The C++ language defines several headers, which contain information that is either necessary or useful to program. For this program, the header <iostream> is included to perform input/output operations. It tells the preprocessor to include the contents of iostream header file in the program before compilation. This file is required for input output statements.*
2. *The line* **using namespace std***; tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++. In C++, namespace defines the scope of the identifiers. A C ++ namespace, contains classes, variables, constants, functions, etc. For using the identifier defined in the namespace scope we must include the using directive.*

   Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. Using and namespace are the new keyword of C++.
3. *The next line* ***// main() is where program execution begins.*** *is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line. For multiline comments we can still use /*.... */.*
4. *The line* **int main()** *is the main function where program execution begins.*
5. *The statement* **cout << "My first C++ program";** causes the string in quotation marks (“ ”) to be displayed on the screen. *This statement introduces two new C++ features,* ***cout*** *and <<. The identifier* **cout** *is a predefined object that represents the standard output stream in C++(i.e., monitor).*

The << operator is called **insertion or put to** *operator and directs (inserts or sends) the contents of the variable/operand on its right to the object on its left. The << operator directs the string constant* "My first C++ program" *to* **cout**, *which sends it for the display to the monitor.*



6. *The next line* **return 0***; terminates main( )function. In C++ main() returns an integer type value to the operating system.*

2.2 Input using cin

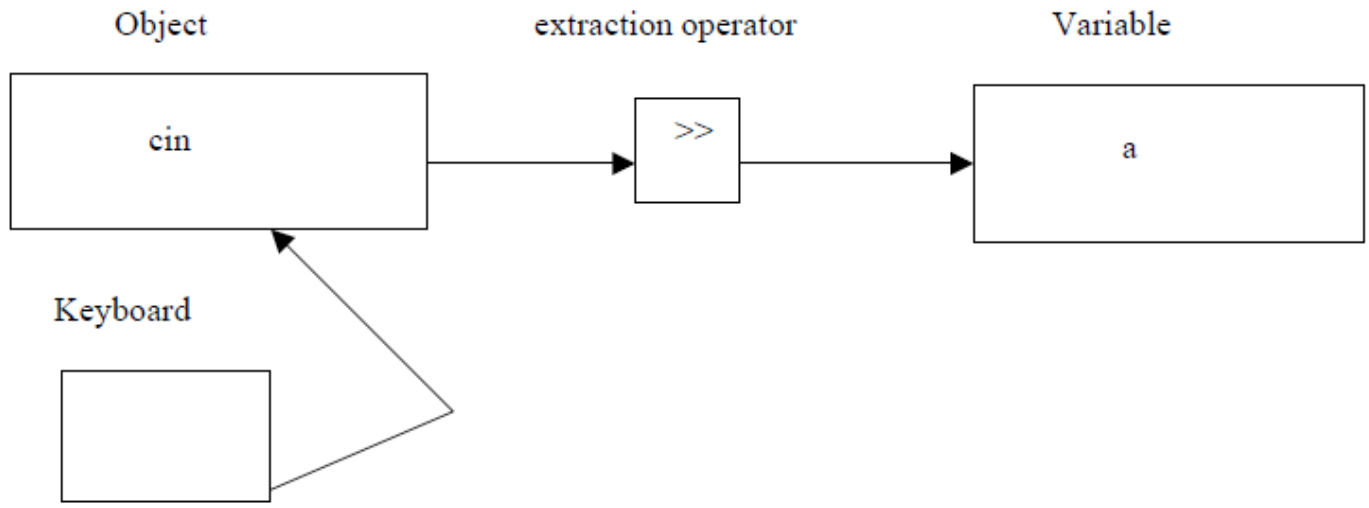
```
#include <iostream>
using namespace std;
int main()
{
        float a, b, sum, avg;
        cout << "Enter two numbers: ";
        cin >> a; //Read numbers
        cin >> b; //from keyboard
        sum = a + b;
        avg = sum/2;
        cout << “Sum: “ << sum <<”\n” ;
        cout<< “Average: “ << avg ;
        return 0;
}
```

The keyword cin (pronounced ‘C in’) is also a stream object, predefined in C++ to correspond to the standard input stream (i.e keyword). This stream represents data coming from the keyboard.

cin >> a;

Is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable a. The identifier cin is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as **extraction or get from** operator and extracts (takes) the value from the stream object **cin** and places it in the variable on its right. For example, in the statement **cin>>a;** the >> operator extracts the value from **cin** object that is entered from the keyboard and assigns it to the variable **a**.



2.3 Cascading of I/O operators

The multiple use of **insertion (<<),** or **extraction (>>)** operator in one statement is called cascading of I/O operators.

For example:

```
cout<< "sum:  " << sum <<endl ;
cin>> a >> b;
```

2.4 Manipulators

Manipulators are the operators/C++ programming language construct used with the insertion operator (<<) to modify or format the data display. The most commonly used manipulators are ***endl, setw, and setprecision***.

2.4.1 The endl Manipulator

This manipulator causes a linefeed to be inserted into the output stream. It has the same effect as using the newline character "\n".

For example:

```
cout<< "Sum: " << sum <<endl ;
cout<< "Average:  "<< avg ;
```

This is equivalent to cout<< "Sum: " << sum <<endl << "Average: "<< avg ;

2.4.2 The setw Manipulator

The manipulator s**etw(n)** specifies a field width n for printing the value. The output is right justified within the field. It is defined inside the header file <*iomanip*>.

For example:

```
cout<<setw(11)<<"Kantipur"<<endl<<setw(11)<<"Engineering"<<endl<<setw(11)<<"College";
```

Output:

```
    Kantipur
 Engineering
     College
```

2.4.3 The setprecision Manipulator

To control the precision of floating-point numbers appearing in the output, setprecision(n) and fixed manipulators are used.

It sets the total number of digits to be displayed witth necessary round-offs, when floating point numbers are used.

setprecision(n), means we are setting n significant digits with necessary round-offs for output. **setprecision** is defined inside the header file <*iomanip*>.

For example:

```
float f =3.14669;
 cout << setprecision(2)<< f << endl;
 cout << setprecision(3) << f << endl;
 cout<<fixed << setprecision(3) << f;
```

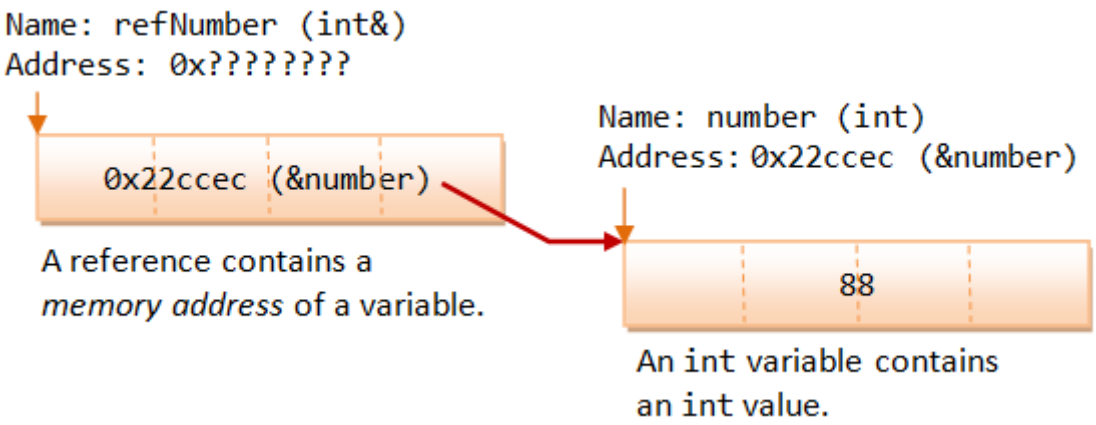
Output:

```
3.1
3.15
3.147
```

2.5 Reference Variable

C++ introduces a new kind of variable known as *reference variable*. *Reference variable* is an alias (another name) given to the already existing variable. When we declare a *reference variable*, it refers to the memory of another variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

C++ references allow you to create a second name for the variable that you can use to read or modify the original data stored in that variable.



A reference must be initialized when it is created. You need to initialize the reference variable during declaration. Once a reference is established to a variable, you cannot change the reference to reference another variable. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

Once a reference is initialized to an object/variable, it cannot be changed to refer to another object/variable. Pointers can be pointed to another object at any time. Pointers can be initialized at any time.

Syntax:

data-type & reference_variable_name = variable_name;

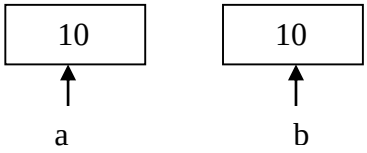
Example:

```
int  p = 100;
int & b = p;     // b is an integer reference to the variable p.
```

Here **p** is an ***integer*** type variable that has already been declared, **b** is the alternative name given to represent the variable **p**.

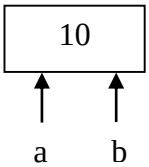
Consider the case of normal variables;

```
int a = 10;
int b = a;
++a;
cout<< a<<b;        // output will be 11 10.
```



Consider the case of reference variables;

```
int a=10;
int &b=a;
++a;
cout<<a<<b;   // output will be 11 11.
```



// what will be the output of the following program? Why?

```
#include<iostream>
using namespace std;
void f( int &);                    //function prototype, one reference variable as argument.
int main()
{
   int s = 4;
   cout<<"s: "<< s<<endl;
   f(s);                           //function call, initialization, int & a = s;
   cout<<"s: "<< s<<endl;
   return 0;
}
void f( int & a)                   //function definition.
{
   a = 10*a;
}
```

Exercise:

Write a function using reference variable as arguments to swap the values of a pair of integers.

2.6 Scope Resolution Operator (::)

Scope resolution operator (::) is used to access the global version of a particular variable. It is also used to declare global variable at local place.

Syntax:

```
        :: variable-name;
```

// Sample program code.

```
char c = 'a';     // global variable
int main()
{
        char c = 'b';                  // local variable with the same name,, local to main.
        cout << "Local c: " << c << endl;       //Displays a character b.
        cout << "Global c: " << ::c << "\n";    // Accesses the global version. Displays a character a.
        ::c ='u' ;          // re-declaration of a global variable. Change persists within this scope
         cout << "Global c: " << ::c << endl;             //Now, displays a character u.
         return 0;
}
```

Suppose in a C program there are two variables with the same name a. Assume that one has become declared outside all functions (global) and another is declared locally inside a function. Now, if we attempt to access the variable a in the function we always access the local variable. This is because the rule says that whenever there is a conflict between a local and a global variable, local variable gets the priority. C++ allows you the flexibility of accessing both the variables. It achieves through a scope resolution operator (: :).

2.7 Specifying Symbolic Constants in C++

A **constant** is an identifier with an associated value which cannot be altered by the program during execution.

1. Using const keyword

```
const int size = 30;
const char n = 'b';
const float PI = 3.14;
```

2. Using #define preprocessor directive

```
#define size 30
#define PI 3.14
```

2.8 String in C++

An ordered sequence of characters is called string. C++ provides following two types of string representations:

2.8.1 The C-style character string

A one-dimensional array of characters which is terminated by a null character '\0'.

Example:

- char address[14];
- char name[] = " Harry";
- char msg[5] = "Hello"; // error in C++.
- char msg[6] = "Hello"; // o.k.

In C++, the size should be one larger than the number of characters in the string. Built-in functions *strlen(), strcpy(), strcmp()* etc. are used to manipulate C-style string.

Reading Embedded Blanks

1. Reading a Line of Text

Input stream object "cin" is used to read a word from keyboard. It cannot read a line of text with embedded blanks. It stops reading when first whitespace is encountered. To read text containing blanks, ***get()*** function is used.

Syntax:

```
cin.get(variable_name, max characters);
```

Example:

```
#include<iostream>
```

```
#include<cstring>
using namespace std;
int main()
{
        char name[70];
        cout<< "Enter your name: " <<endl;
        cin.get(name,70);                    //can read one line of text with whitespaces, maximum 70.
        cout<<"Your name is: "<<name;
        return 0;
}
```

2. Reading Multiple Line of Text

To read multiple line of text, another form of ***get( )*** function is used.

Syntax:

cin.get (variable_name, max, 'terminator symbol');

Example:

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{       const int max = 80;
        char poem[max];
        cout<< "enter multiple lines of text: " <<endl;
        cin.get(poem, max, '$');             //can read multiple lines of text, maximum characters 80.
        cout<<"You have entered following lines of text: "<<poem;
        return 0;
}
```

In this example, ***get()*** function continue to accept characters until terminating character '$' or until we exceed the size of max.

2.8.2 The string class type introduced with Standard C++

The standard C++ library provides a string class, which reduces several problems related to size, operations etc. with C-style strings. For this we need to include <string> header file.

- **Example:**

```
string str;                    // Declaring a C++ string object:
str = " Hello";                // Initializing a C++ string object:
string str1 = "Send money!";   // Another way.
```

- **Sample program code:**

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  string str1 = "Hello";
  string str2 = "World";
  string str3, str4;
  str3 = str1;              // copy str1 into str3.
  cout << "str3 : " << str3 << endl;
  str4 = str1 + str2;
  cout << "str4:  " << str4 << endl;
  cout<<"Size of string 3:  "<<str3.size()<<endl;      //ReturnS length of string str3 in bytes.
  return 0;
}
```

Another Program: This program shows the use of getline () function.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
  string str1, str2;
  cout<<" Enter a line of text: "<<endl;
  getline(cin, str2);              //to read a line of text with blanks.
  cout<<"Your line of text is: "<<endl<<str2<<endl;
  cout<<" Enter multiple lines of text: "<<endl;
  getline(cin, str1, '$');   //to read multiple lines of text with embedded blanks, $ is the terminator character.
  cout<<"You entered following line of text"<<endl<<str1<<"  Length is "<<str1.size()<<endl;
  string text="jhskfshfs ihfihsifhwsihfi wsh";        //string constant
  cout<<endl<<"Given string constannt: "<<text<<endl;
   return 0;
}
```

TRY this program, and ANALYZE the output.

2.9 Type Conversion

Type conversion refers to changing an entity of one data type, expression, function argument, or return value into another. There are two types of type conversion: automatic conversion and explicit type conversion.

2.9.1 Automatic Conversion (Implicit Type Conversion)

Implicit type conversion, also known as *coercion*, is an automatic type conversion done by the compiler.

Example:

```
int a = 5.6;
cout<< "a: " <<a;
```

OUTPUT:

a: 5

Whenever the compiler expects data of a particular type, but the data is given as a different type, it will try to automatically covert. For e.g.

int a=5.6 ;

float b=7 ;

In the example above, in the first case an expression of type float is given and automatically interpreted as an integer. In the second case, an integer is given and automatically interpreted as a float. There are two types of standard conversion of numeric type: promotion and demotion. Demotion is not normally used in C++ community.

Promotion:

Promotion occurs whenever a variable or expression of a smaller type is converted to a larger type.

```
float a=4 ;      //4 is a int constant, gets promoted to float
long b=7 ;       // 7 is an int constant, gets promoted to long
double c=a ;     //a is a float, gets promoted to double
```

When two or more operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is also called type promotion.

Example:

```
int a = 5;
float b = 7.6;
cout<< "sum: " << a+b;        //sum will be 12.6,
```

WHY?

Analyze the output of the following program:

```
float c=13.2;
char p='Z';
cout<<c+p;          //What is the output? WHY?
```

There is generally no problem with automatic promotion. Programmers should just be aware that it happens.

Demotion:

Demotion occurs whenever a variable or expression of a larger type gets converted to smaller type. By default, a floating point number is considered as a double number in C++.

```
int a=7.5       // double gets down – converted to int ;
int b=7.45;     // float gets down – converted to int
char c=b ;      // int gets down – converted to char
```

Standard Automatic demotion can result in the loss of information. In the first example the variable 'a' will contain the value 7, since int variable cannot handle floating point values.

The order of types is given below:

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

2.9.2 Explicit Type Conversion (Type Casting)

Explicit type conversion is a type conversion which is explicitly defined within a program by the programmer. Compile is not responsible for this type of conversion.

Syntax:

```
(type-name) expression          //C notation
 type-name (expression)         //C++ notation
```

Example:

```
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result = int (da )+ int (db) + int (dc);        //result = 9
cout<< "Result: " <<result;
```

Another Example:

```
int p =100;
cout<< char( p)<<"\n";
```

Explicit type conversion from *int* type to *char* type.

//output will be d.

//WAP that inputs a character from keyboard and displays its corresponding ASCII value on the screen.

```
#include<iostream>
using namespace std;
int main()
{
        char d;
        cout<< "Enter any character: "<<endl;
        cin>>d;
        cout<<"ASCII value of "<<d<<" is "<< int(d);        //type conversion, char type to int type.
        return 0;
}
```

Analyze the output of the following program:

```
#include<iostream>
using namespace std;
int main()
{
         float a=3.4, b=4.9, c=13.2;
        int d=a+b+c;
        cout<<d<<endl;          //output is 21, WHY?
        int e= int(a)+int (b)+ int (c);
        cout<<e<<endl;                  //output is 20, WHY?
        return 0;
}
```

Chapter 3

Functions in C++

3. 1 Introduction

A function is a named unit of a group of program statements. The unit can be invoked from other part of the program. Dividing a program into function is one of the major principles of top-down structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

```
void show( ) ;          / * Function declaration * /
int main ( )
{
- - - - - -
- - - - - -
show( ) ;               / * Function call * /
}
void show( )            / * Function definition * /
{
- - - - - -
- - - - - -              / * Function body * /
- - - - - -
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and controls return to the main program when the closing brace is encountered. C++ has added many new features to functions to make them more reliable and flexible.

3. 2 C++ User-defined Function Types

1. **Function with no argument and no return value**

```
#include<iostream>
using namespace std;
void msg()  //eliminating the function declaration(function definition preceding function call).
{
     cout<<" true beauty ";
}
int main()
{
   msg();   //function call.
   return 0; }
```

2. **Function with no argument but return value**

```
#include<iostream>
using namespace std;
int area();
int main()
{
      cout<<"Area: "<<area();
      return 0;
}
int area()
    {
          int l, b;
          cout<< "Enter l, and  b: "<<endl;
          cin>>l>>b;
          return l*b;
    }
```

3. **Function with argument but no return value**
   - void area(int, int); //function declaration.
4. **Function with argument and return value.**
   - float area(float, int); //function declaration.

3.3 Passing Arguments to Function

An argument is a data passed from a program to the function. In function, we can pass a variable by three ways:

1. **Passing by value**

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameters are not changed.

```
#include<iostream>
using namespace std;
void swap(int, int) ;
int main ( )
{
   int x, y ;
   x=10 ;
   y=20 ;
   swap(x, y) ;
   cout << "x = "<<x<<endl ;
```

```
    cout<< "y = " <<y<<endl ;
    return 0;
}
```

```
void swap(int a, int b) // function definition
{
    int t ;
    t=a ;
    a=b ;
    b=t ;
}
```

Output:

```
    x = 10
    y = 20
```

2. Passing by reference

Passing argument by reference uses a different approach. In this, the reference of original variable is passed to function. Formal parameters become references (aliases) to the actual parameters in the calling function. The main advantage of passing by reference is that the function can access the actual variable in the calling program. The second advantage is this provides a mechanism for returning more than one value from the function back to the calling program.

Example:

```
void swap(int &, int &) ;          //reference variables as arguments.
int main ( )
{
    int x, y ;
    x=10 ;
    y=20 ;
    swap(x, y) ;                   //Function call.
    cout << "x = "<<x<<endl ;
    cout<< "y = " <<y<<endl ;
    return 0;
}
```

```
void swap(int & a, int & b)          // function definition.
{
   int t ;
   t=a ;
   a=b ;
   b=t ;
}
```

Output:

x=20
y=10

3. Passing by address or pointer

This is similar to passing by reference but only difference is in this case, we can pass the address of a variable.

```
#include<iostream>
using namespace std;
void swap(int *, int *) ;
int main ( )
{
   int x, y ;
   x=10 ;
   y=20 ;
   swap(&x, &y) ;  //Passing addresses of actual parameters.
   cout << "x = "<<x<<endl ;
   cout<< "y = " <<y<<endl ;
   return 0;
}
```

```
void swap(int *a, int *b) // function definition
{
   int t ;
    t= *a;
   *a= *b;
   *b= t;
}
```

Output:

x=20

3.4 Function Overloading

Function overloading (also method overloading) is a programming concept that allows programmers to define two or more functions with the same name and in the same scope.

Each function has a unique signature (or header), which is derived from:

- function/procedure name
- number of arguments
- arguments' type
- arguments' order
- arguments' name

Please note: Not all above signature options are available in all programming languages.

Two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs or both. When two more functions share the same name but with different parameters, different number of parameters, or both, they are said overloaded.

Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and / or type of arguments used to call the function.

```
// Program illustrate function overloading. // Function area( ) is overloaded.
#include <iostream>
int area(int) ;
double area(double, int) ;                    // Declarations (prototypes)
long area(long, int, int) ;
int main( )
{
        cout<<area(10)<< endl ;
        cout<<area(2.5,8)<< endl ;
        cout<<area(100, 75, 15)<< endl ;
        return 0 ;
}
int area(int s)                    // square
{
        return(s*s);
}
double area(double r,  int h)   // Surface area of cylinder ;
{
        return(2*3.14*r*h) ;
}
long area(long l,  int b,  int h)          //area of parallelepiped
{
        return(2*(l*b+b*h+l*h)) ;
}
```

Output:

100

125.6

20250

3.5 Default Arguments

In C++ programming, you can provide default values for function parameters. The idea behind default argument is simple. If a function is called by passing argument/s, those arguments are used by the function. But if the argument/s are not passed while invoking a function then, the default values are used. Default value/s are passed to argument/s in the function prototype.

C++ allows us to assign default value(s) to a function's parameters, which are called default arguments. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. *If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.*

Example:

```
void f(int i, int j=7) ;                    // legal
void g(int i=3, int j) ;                    // illegal
void h(int i, int j=3,int k=7) ;            // legal
void m(int i=1, int j=2, int k=3) ;         // legal
void n(int i=2, int j, int k=3) ;           // illegal, why?
```

Remember: *Only the trailing arguments can have default values i.e., that is we must add default values from right to left.*

SAMPLE PROGRAM:

```
#include <iostream>
using namespace std;
int divide (int , int =2);          // function declaration; with one default argument.
int main ()
{
    cout << divide (12);            //default for 2nd argument.
    cout << endl;
    cout << divide (20,4);          //pass all arguments explicitly.
    return 0;
}
```

```
int divide (int a, int b)          //function definition
{
        int r;
        r=a/b;
        return (r);
}
```

Note:

The default values are specified at the time of function declaration.

Any argument in a function cannot have a default value unless all arguments appearing on its right have their default values.

If a function has N default arguments then it can be invoked N+1 different ways.

Another Way:

```
#include<iostream>
using namespace std;
int divide (int a , int f=2)          // function prototype; with one default argument.
{
         return a/f;
}
int main ()
{
        cout << divide (12);          //default for 2nd argument.
        cout << endl;
        cout << divide (20, 4);              //pass all arguments explicitly.
        return 0;
}
```

Advantages of providing default arguments:

1. Default arguments are useful when some arguments always have the same value.
2. We can use default arguments to add new parameters to the existing functions.
3. Default arguments can be used to combine similar functions into one.

3.6 Inline Functions

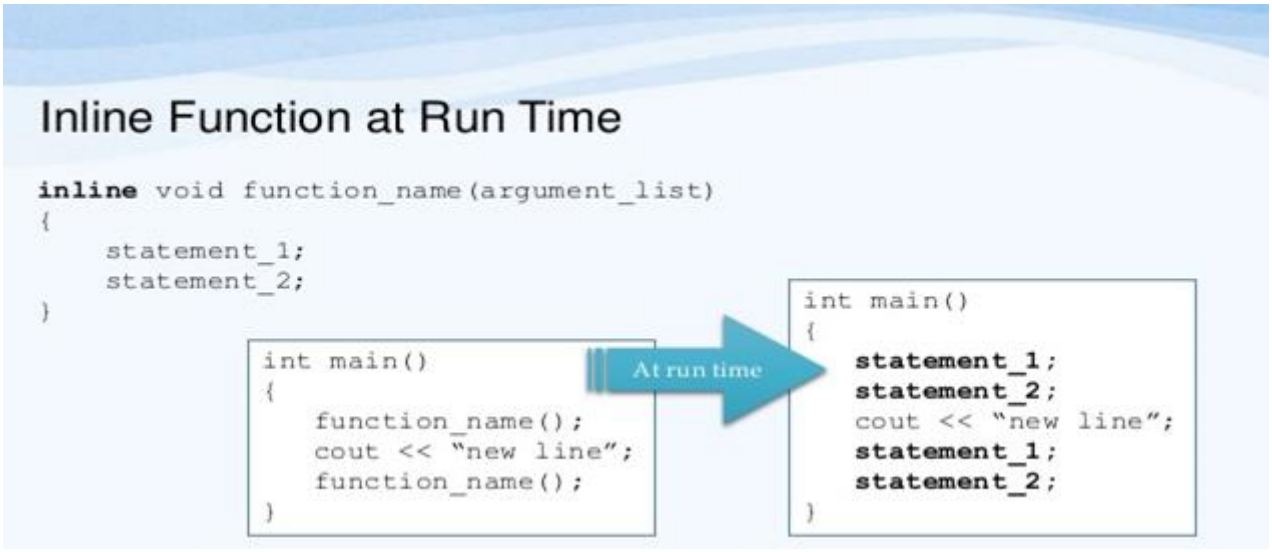
Function call is a costly operation. During the function call it's execution take overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading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

The inline functions are a C++ enhancement feature to increase the execution time of a program. Inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function body.

Inline functions are similar to macros, but macros are not functions. And macros are treated differently by the compiler.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

The compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.



The advantage of inline functions is that they can be executed much faster than normal functions. The disadvantage of inline functions is that it takes more memory than normal function. In general, only short functions are declared as inline functions.

To in-line a function, simply precede the function's definition with the inline keyword.

Syntax:

inline return_type function_name(arguments)

{

//function body.

}

EXAMPLE:

```
#include <iostream>
using namespace std;
inline int sum(int a, int b)
{
        return a+b;
}
```

```
int main()
{
        int x, y, s;
        cout<<"Enter two numbers:"<<endl;
        cin>>x>>y
        s=sum(x, y);                    //function call.
        cout<< "Sum= "<<s<<endl;
}
```

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

Pros:-

1. It speeds up your program by avoiding function calling overhead.
2. It save overhead of variables push/pop on the stack, when function calling happens.
3. It save overhead of return call from a function.
4. It increases locality of reference by utilizing instruction cache.
5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)

Cons:-

1. It increases the executable size due to code expansion.
2. C++ inlining is resolved at compile time. Which means if you change the code of the inlined function, you would need to recompile all the code using it to make sure it will be updated
3. When used in a header, it makes your header file larger with information which users don't care.

3.7 Math library functions

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the #include directive at the top of the program file, e.g. for common mathematical calculations we include the file cmath with the #include<cmath> directive which contains the *function prototypes* for the mathematical functions in the cmath library.

Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

Function	*Description*
sqrt(x)	square root
sin(x)	trigonometric sine of x (in radians)
cos(x)	trigonometric cosine of x (in radians)
tan(x)	trigonometric tangent of x (in radians)
exp(x)	exponential function
log(x)	natural logarithm of x (base e)
log10(x)	logarithm of x to base 10
fabs(x)	absolute value (unsigned)
ceil(x)	rounds x up to nearest integer
floor(x)	rounds x down to nearest integer
pow(x, y)	x raised to power y

Sample Program:

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
   int a,b, c;
   float d=45.6,f=78.9;
   cout<<"enter values of a & b: " <<endl;
   cin>>a>>b;
   c = pow(a, b);
   cout<<b<<" Power of "<<a<<": "<<c<<endl;
        cout<<"Square root of "<<a<<": "<<sqrt(a)<<endl;;
         cout<<"Ceiling of "<<d<<": "<<ceil(d)<<endl;
        cout<<"Floor of "<<d<<": "<<floor(d)<<endl;
        return 0;
}
```

3.8 Return by Reference

When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structure objects, and class objects.

When a function returns a reference, it returns an address of returned variable. This way, a function can be used on the left side of an assignment statement. Consider the following example.

```
#include <iostream>
using namespace std;
int num;          // Global variable
int& test();      // Function declaration
int main()
{
   test() = 5;              //assigns 5 to num, here function is on the left side of the assignment operator.
   cout << num;
   return 0;
}
int& test()
{
   return num;              //address of num is returned.
}
```

Output

```
5                 //WHY?
```

In program above, the return type of function test() is int&. Hence, this function returns a reference of the variable num.

The return statement is

```
return num;
```

Unlike return by value, this statement doesn't return value of num, instead it returns the variable itself (address).

So, when the variable is returned, it can be assigned a value as done in

```
test() = 5;
```

This stores 5 to the variable num, which is displayed onto the screen.

Consider another example:

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

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i )
{
        return vals[i];          // returns a reference to the ith element
}
int main ()                              // main function to call above defined function.
{
  cout << "Value before change" << endl;
  for ( int i = 0; i < 5; i++ )
        {
                cout << "vals[" << i << "] = ";
                cout << vals[i] << endl;
        }

  setValues(1) = 20.23;          // changes value at index 1.
  setValues(3) = 70.8;           // changes value at index 3.

  cout << "Value after change" << endl;
  for ( int i = 0; i < 5; i++ )
        {
                cout << "vals[" << i << "] = ";
                cout << vals[i] << endl;
        }
  return 0;
}
```

Output:

```
Value before change
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1

Value after change
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
```

vals[3] = 24.1

vals[4] = 50

Important Things to Remember When Returning by Reference

1. Ordinary function returns value but this function doesn't. Hence, you should not return a constant from the function.

```
int& test()
{
        return 2;
}
```

2. You should not return a local variable from this function.

```
int& test()
{
         int n = 2;
        return n;
}// value is destroyed here, because n is local variable.
```

3. **The variable whose reference is being returned may only be:**

- Global variable
- Static variable
- Caller itself (Pass by reference or address).

3.9 Storage Classes

- A storage class defines the ***scope*** (visibility) and ***life-time*** of variables and/or functions within a C++ Program. How storage is allocated for variables and how variable is treated by complier depends on these storage classes.
- The scope of the variable determines which parts of the program can access it.
- And the lifetime refers how long it stays in existence. That is, the duration till which a variables remains active during program execution.
- There are four storage classes:
  - **auto**
  - **register**
  - **static**
  - **extern or global**

3.9.1 The auto Storage Class

- The **auto** storage class is the default storage class for all local variables.
- Its visibility is restricted to the function in which it is declared. Its lifetime is also limited to till the time its container function (function in which they are declared) executing.
- **Example:**

```
void somefunc()
{
        int count;              //by default auto
        auto int month;
}
```

- The example above defines two variables with the same storage class; auto can only be used within functions, i.e., local variables.

3.9.2 The register Storage Class

- Similar in behavior(*visibility & lifetime*) to an automatic variable, except how they are stored in the memory. automatic variables are stored in the primary memory but register variables are stored in CPU register.
- The objective of the register variable is to increase the access speed to execute the program faster.
- **Example:**

```
{
    register int  length;
}
```

3.9.3 The extern Storage Class

- Variables that are declared outside of any function are called External or global variables. These variables have external storage class.
- A global variables are visible to all the functions present in the program i.e., global scope.
- The lifetime of a global or external variable is same as the lifetime of a program.
- **Example:**

```
int a=20;       // global or external variable
void somefunc()
{
        int x;
        x = a+20;
}
```

3.9.4 The static Storage Class

- Variables which are declared using the keyword static are called static variables.
- A static variable has the visibility of a local variable but the lifetime of the external variable.
- **Example:**

```
void func(void);
static int count = 15;          /* Global static variable */
int main()
{
        do
          {
              func();
              count--;
           } while(count > 10);
          return 0;
}
void func( void )                    // Function definition
{
        static int i = 5;          // local static variable
          i++;
         cout << "i is " << i ;
         cout << " and count is " << count << endl;
}
```

Summary:

	Automatic	External	Static	Register
Lifetime	Function block	Entire program	Entire program	Function block
Visibility	Local	Global	Local	Local
Initial Value	Garbage	0	0	Garbage
Storage	Stack segment	Data segment	Data segment	CPU Registers
Purpose	Local variables used by a single function	Global variables used throughout the program	Local variables retaining their values through-out the program	Variables using CPU registers for storage purpose
Keyword	auto	extern	static	Register

Important Questions

1. Discuss the feature of Object-Oriented Programming? Differentiate between Object-Oriented Programming & Procedural Based Programming.
2. What is type casting? Explain with suitable example.
3. Write a program to find the cube of given integer using inline function.
4. Write a function using reference variables as arguments to swap the values of a pair of integers.
5. Write a program that inputs a character from keyboard and displays its corresponding ASCII value on the screen.
6. What is function overloading? Explain with suitable example.
7. Write a program to compute the area of a triangle, a rectangle, square, and a circle by overloading the area() function.
8. Explain the inline function with example.
9. WAP to find the square of given integer using inline function.
10. Write a program to find the square root of given integer using inline function.
11. Explain the use of default arguments with suitable example.
12. Explain the do/while structure.
13. Explain the use of break and continue statements in switch case statements in C++
14. Differentiate between function and macro with suitable examples.
15. Differentiate between user defined functions and library functions with suitable examples.
16. Explain different storage classes of C++.

Chapter 4

Structure in C++

Introduction

A *data structure* is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths.

Data structures can be declared in C++ using the following syntax:

```
struct type_name
{
      member_type1 member_name1;
      member_type2 member_name2;
      member_type3 member_name3;
      .
      .
} object_names;
```

Where `type_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{}`, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

For example:

```
struct product {
  int weight;
  double price;
} ;

```

This declares a structure type, called `product`, and defines it having two members: `weight` and `price`, each of a different fundamental type. This declaration creates a new type (`product`), which is then used to declare three objects (variables) of this type: `apple`, `banana`, and `melon`.

Right at the end of the `struct` definition, and before the ending semicolon (`;`), the optional field `object_names` can be used to directly declare objects of the structure type.

For example, the structure objects `apple`, `banana`, and `melon` can be declared at the moment the data structure type is defined:

```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

Declaring object's/instance/variables of a structure

Once structure is defined, we can create variables of type structure.

Syntax:

Structure_name variable_name;

Example:

product apple; //creating an instance of product.

product banana, melon; //creating two instances of product.

Accessing Structure Members

To access any member of a structure from, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.

Syntax: structure_variable_name.member_name;

Example: apple.weight

apple.price

banana.weight

banana.price

Complete Example

```
#include <iostream>
using namespace std;
struct Person
{
   char name[50];
   int age;
   float salary;
};
int main()
{
   Person p1;
   cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
```

```
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;

    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;
    return 0;
}
```

Some important differences between the C and C++ structures:

1. Member functions inside structure: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.

2. Direct Initialization: We cannot directly initialize structure data members in C but we can do it in C++.

```
// CPP program to initialize data member in c++
#include <iostream>
using namespace std;
  struct Record {
    int x = 7;                    //ok in C++; not allowed in C.
};
// Driver Program
int main()
{
    Record s;
    cout << s.x << endl;
    return 0;
}
// Output
// 7
```

3. Using struct keyword: In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record.

In C, we must use "struct Record" for Record variables. In C++, we need not use struct and using 'Record'only would work.

4. Data Hiding: C structures do not allow concept of Data hiding but is permitted in C++.

5. Inheritance: C++ structures support inheritance but C structure does not i.e. we can derive new structure from already defined structure.

6. Access Modifiers: C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

Example:

```
struct EMPLOYEE                    //defining a structure EMPLOYEE.
{
   string Name;       //By Default public, so can be accessed from main().
private:              //Can't be accessed from main().
   int Id;
   float Salary;
protected:            //Can't be accessed from main().
   string Address;
   string Company;
public:               //public access specifier so Can be accessed from main().
   void getdata()
   {
      cout<<"Enter Name,Id,Salary,Address, and Company Name: "<<endl;
      cin>>Name>>Id>>Salary>>Address>>Company;
   }
   void display();       //Declaration.
```

```
void Calc_Tax()
   {
      if(Salary<20000)
         cout<<"Tax amount is: "<<Salary*0.05<<endl;
      else
         cout<<"Tax amount is: "<<Salary*0.1<<endl;
   }
};
void EMPLOYEE::display()          //Definition outside the structure so membership label is mandatory.
{
   cout<<"The Employee Details are: "<<endl;
   cout<<"Name: "<<Name<<endl<<"Id: "<<Id<<endl;
   cout<<"Address: "<<Address<<endl<<"Salary: "<<Salary<<endl;
   cout<<"Company: "<<Company<<endl;
}
int main()
{
   EMPLOYEE E, S;      //creating two instances of EMPLOYEE
   E.getdata();
   E.display();
   E.Calc_Tax();
   S.getdata();
   S.display();
   S.Calc_Tax();
   return 0;
}
```

Remember: You can also use ***new*** and ***delete*** operator with structure. Syntax is similar to built-in type.

```
EMPLOYEE *Z= new EMPLOYEE; //creates a variable of EMPLOYEE and assigns its address to
                           EMPLOYEE pointer Z.
Z->getdata();              //once structure pointer is defined, structure members are accessed using ->
                           (arrow operator)
```

```
        Z->display();
        Z->Calc_Tax();

        delete Z;                    //releasing memory using delete
```

What should be changed in above program if we want to enter data of 10 employee???

THINK!!!

Analyze the following program.

```
#include<iostream>
using namespace std;
struct EMPLOYEE
{
   string Name;
private:
   int Id;
   float Salary;
protected:
   string Address;
   string Company;
public:
   void getdata()
   {
      cout<<"Enter Name,Id,Salary,Address, and Company Name: "<<endl;
      cin>>Name>>Id>>Salary>>Address>>Company;
   }
   void display();
};
```

```
void EMPLOYEE::display()
{
   cout<<"The Employee Details are: "<<endl;
   cout<<"Name: "<<Name<<endl<<" Id: "<<Id<<endl;
   cout<<"Address: "<<Address<<" Salary: "<<Salary<<endl;
   cout<<"Company: "<<Company<<endl;
}
int main()
{
   EMPLOYEE E[10];                    //An array of 10 EMPLOYEE.
   for(int i=0; i<10; i++)
   {
      E[i].getdata();
   }
   for(int i=0; i<10; i++)
   {
      E[i].display();
   }
}
```

Chapter 5
Class and Objects

1. Introduction

In OOP data and functions are wrapped up into a single unit, which is called class. In other words, class is a unit that contains a group of logically related data items and functions that work on them. Once a class is defined, we can declare variables of that type i.e., we can create objects. A class type variable is called an object or instance. A class would be the data type, and an object would be the variable. Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use. Classes are also referred to as programmer-defined data types.

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

The variables and functions defined inside the class are called ***class members***. Variables declared inside the class are called ***data members*** and functions inside the class are called ***member functions***.

A class definition starts with the keyword ***class*** followed by the ***class name***; and the ***class body***, enclosed by a pair of curly braces. A class definition must be followed by a semicolon. Classes are generally declared with the following format:

```
class class_name
 {
   access-specifier:
          variable declarations;
          function declarations;
           …….
   access-specifier:
          variable declarations;
          function declarations;
          ……
   access-specifier:
         variable declarations;
         function declarations;
 };       //end of class
```

Example:

```
class Room
{
        int length;
        int breadth;
};
```

2. Visibility labels or Access Specifiers or Encapsulation Modifiers

The members of a class are classified into three categories: *private, public, and protected*. *Private*, *protected*, and *public* are reserved words and are called member *access specifiers* or *visibility labels*. There are 3 types of access specifiers or visibility modifiers. They are;

1. *public*
2. *protected*
3. *private.*

By default class members are private. If all the three visibility labels are absent, then by default, all the members are private.

private members of a class are accessible only from within the same class using member functions. We cannot access them outside of the class. That is only the member functions can have access to the private members (both data & functions).

protected members are accessible with member functions of the same class and also from any class immediately derived from it (i.e. immediate subclasses).

public members are accessible from anywhere where the object is visible. That is, public members (both data & functions) can be accessed from outside the class.

Example 1:

```
class student
 {
  private:
        int roll;
        float marks;
```

```
public:          //Remember member functions are public, so can be accessed from main()
        void getdata(int a, float b)          // function definition.
        {
                roll = a;
                marks = b;
        }
        void displaydata()
        {
                cout<<"Roll number : "<<roll<<"\nMarks : "<<marks;
        }
 };
int main()
{
        student ram;              // here, ram is an object of type student.
        ram.getdata(12,56);       // function call using object ram, values 12 and 56 are assigned to  data
                                  members roll and marks.
        ram.displaydata()         // another function call to display data.
        return 0;
}
```

In this example functions are defined inside the class.

C++ Structure and Class:

By default, the members of a class are private, while, by default, the members of a structure are public. Following simple example represents this fact.

```
class student
 {
        int roll;
        float marks;
 };
int main()
{
        student Ram;
        Ram.roll = 10;          //not allowed, private
        Ram.marks =150;         //not allowed, private
}
```

```
struct student
 {
        int roll;
        float marks;
 };
int main()
{
        student Ram;   //C++ declaration, but error in C.
        Ram.roll = 10;          //allowed, public
        Ram.marks =150;         //allowed, public
}
```

3. Data Hiding and Encapsulation

The binding of data and functions together into a single class type variable is called data encapsulation. The key feature of object oriented programming is data hiding. The insulation of the data from direct access by the program is data hiding or information hiding.

4. Object Declaration/Define C++ Objects

Once a class is defined, you can declare objects of that type. The syntax for declaring an object is the same as that for declaring any other variable.

Syntax:

class_name object_name;

Example:

The following statements declare two objects of type ***student***:

student Ram, Hari; //Creating two objects Ram and Hari.

5. Accessing Class Members

Once an object of a class is declared, it can access the public members of the class. The private members cannot be accessed directly from outside of the class. The private data of class can be accessed only by the member functions of that class. The public member can be accessed outside the class from the main function. When an object of the class is created then the members are accessed using the '.' dot operator.

Synatx:

for calling a public member data,

object_name.data_member_name;

for calling a public member function,

object_name.function_name (actual_arguments);

6. Scope of Class and its Members

```
class Sample
{
        int x, y ;
        void func()
        {
            cout<<"Hello from private function"<<endl;
        }
    protected:
        int k;
        void PFunc()
```

```
        {
            cout<<"Hello from Protected function"<<endl;
        }
    public:
        int z ;
        void funcall()
        {
            cout<<"Hello from Public function"<<endl;
            func();         //Allowed
            PFunc();        //Allowed

        }
        void setdata(int a, int b, int c, int d)
        {
            x=a;
            y=b;
            k=c;
            z=d;
        }
        void display()
        {
            cout<<"x:"<<x<<endl;
            cout<<"y: "<<y<<endl;
            cout<<"z: "<<z<<endl;
            cout<<"k: "<<k<<endl;
        }
}; //class definition ends here
int main( )
{
   Sample P;
   P.x=0;              // error, x is private.
   P.k=100;     //error, k is protected.
   P.z=10 ;     // ok, z is public.
   cout<<"z: "<<P.z<<endl; //ok
   P.func();    // error, func() is private.
   P.PFunc();   //error, PFunc() is proteceted.
```

```
    P.funcall();  //ok, funcall()is public.
    P.setdata(4, 5, 6, 8);  //ok, setadata is public.
    P.display();            //ok, display is public.
    return 0;
}
```

7. Defining Member function of class

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

7.1 Inside the class definition

❖ When a function is defined inside the class, it is treated as an inline function.

❖ Normally only small functions are defined inside the class.

Example:

```
class Student
{
      int roll;
      char name[30];
   public:
      void getdata(void)
      {
        cout<<" Enter name: ";
        cin>>name;
        cout<<" Roll: ";
        cin>>roll;
      }
      void display()
      {
         cout<<"Name: " <<name<<endl<<"Roll: " <<roll<<endl;
      }
};
```

```
int main()
{
        Student A, B;
        A.getdata();
        A.display();
        B.getdata();
        B.display();
        return 0;
}
```

7.2 Outside the class definition:

- ❖ When function is defined outside the class, function header contains the '*identity label*' or '*membership label'*.
- ❖ This *label* tells the compiler that which class the function belongs to. General form of a member function definition is:

return-type class-name : : function-name (argument declarations)

```
        {
                //function body.
        }
```

The membership label ***class-name ::*** tells the compiler that the function function-name belongs to the class class-name.

Characteristics of member functions:

- Several classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class.
- A member function can call another member function directly, without using the dot operator.

Example 2:

```
class student    //specify a class
 {
  private :
        int rollno;        //class data members
        float marks;
  public:
        void initdata(int r, int m)
        {
                rollno=r;
                marks=m;
        }
    void getdata(); //member function to get data from user
    void showdata();// member function to show data
 };
 void student :: getdata()
 {
    cout<<"Enter Roll Number : ";
    cin>>rollno;
    cout<<"Enter Marks : ";
    cin>>marks;
 }
 void student :: showdata()
 {
    cout<<"Roll number : "<<rollno<<" \nMarks : "<<marks;
 }
int main()
 {
    student st1, st2; //define two objects of class student
    st1.initdata(5,78); //call to member function to initialize
    st1.showdata() ;//call, member function to display data
    st2.getdata(); //call, member function to input data
    st2.showdata(); //call, member function to display data
    return 0;
 }
```

Another Example:

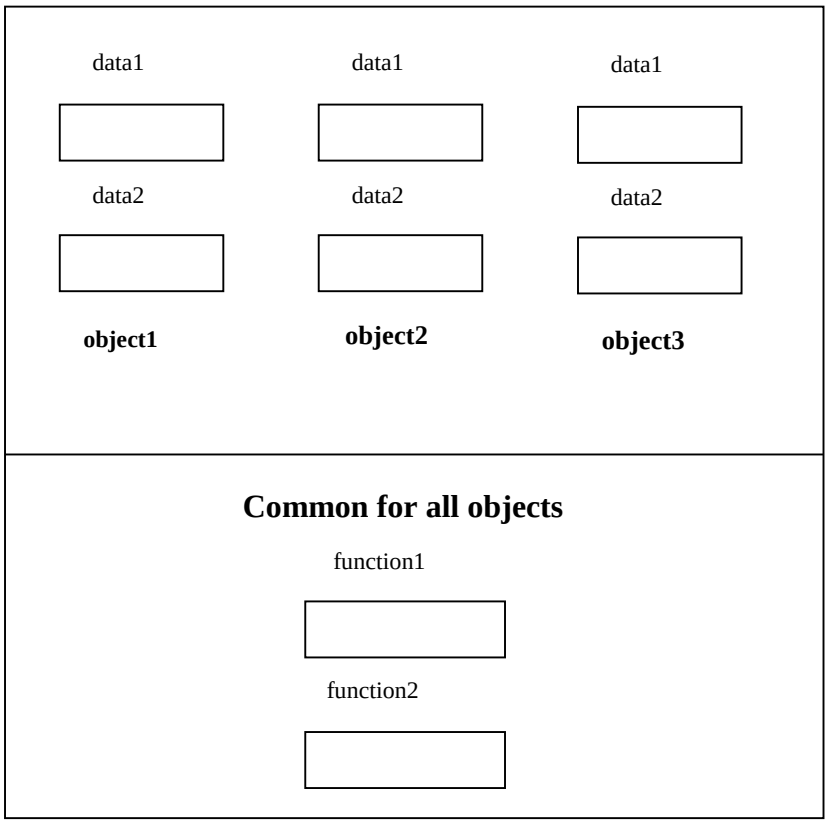
```
#include<iostream>
#include<cstring>
using namespace std;
class Student
{
      int roll;
      char name[30];
   public:
      void getdata(int a, char c[])
      {
         roll= a;
         strcpy(name, c);
      }
      void display()
      {
         cout<<"Roll: " <<roll<<endl<<"Name: " <<name;
      }
};

int main()
{

         char x[30]= "Ram";
         int a= 20;
         Student A;
         A.getdata(a, x);
         A.display();
         return 0;
}
```

Exercise:

1. Define a class **Interest** with following specifications:
    a. Data Members
        i. principal
        ii. time
        iii. rate
    b. Member Functions
        i. getdata() to assign initial values
        ii. display() to display values
        iii. Finterest() to find and display the interest.
2. Define a class **Room** with following specifications:
    a. Data Members
        i. length
        ii. width
    b. Member Functions
        i. getdata() to assign initial values
        ii. display() to display length and width
        iii. area() to find and display the area.
3. Define a class **Employee** with following specifications:
    a. Data Members
        i. name
        ii. age
    b. Member Functions
        i. getdata() to assign initial values
        ii. display() to display name and age
4. Define a class **Account** with following specifications:
    a. Data Members
        i. name of the depositor
        ii. account number
        iii. balance amount in the account
    b. Member Function
        i. To assign initial values
        ii. To deposit an amount
        iii. To withdraw an amount after checking the balance.
        iv. To display name and balance.

8. Memory Allocation for Objects

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



9. Nesting of Member Functions

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions. For example, the class below shows the nesting of member function ***findinterest()*** inside the member function ***findtotal()***.

```
class total
{
        private:
        float principle, time, rate;
        float findinterest()
        {
                return principle * time * rate / 100;
        }
        public:
```

```
        void setdata(float p, float t, float r)
        {
                principle = p;
                time = t;
                rate = r;
        }
        float findtotal()
        {
                return principle + findinterest();
        }
};
```

Remember: Like private data member, some situations may require certain member functions to be hidden from the outside call. In the above example, the member function *findinterest()* is in private block and is hidden from the outside call. A private member function can only be called by another member function that is a member function of its class.

Another Example:

```
class set
{
int m,n;
public:
void input(void);
void display(void);
void largest(void);
};
int set :: largest(void)
{
if(m >= n)
return(m);
else
return(n);
}
void set :: input(void)
{
        cout << "Input value of m and n"<<"\n";
        cin >> m>>n;
}
void set :: display(void)
{
    cout << "largest value=" << largest() <<"\n";
}

int main()
{
 set A;
 A.input();
 A.display();
return 0;
}
```

The output of program would be:

```
Input value of m and n
25 18
Largest value=25
```

10. Static Members of a Class

Class can have ***static data members*** and ***static member functions***. Static variables declared inside the class are called ***static data members*** and member functions that are declared ***static*** are called ***static member functions*** *of the class.*

- For e.g.:

```
class test
{
        static int a;          // static data member a.
        int b;
public:
        static void getdata();  // static member function getdata().
};
```

10.1 Static Data Members

- The static member variables have similar properties to that of C static variables (life time and scope).
- But while declared as a class member static member variable has certain special characteristics as:
  - Static data member is shared between all objects of the class. That is only one copy of that member is created for the entire class no matter how many objects are created.
  - It is visible only within the class, but it has the lifetime of entire program.
  - *Static variable are normally used to maintain values common to the entire program.*
    - *For example, a static data member can be used to count the occurrences of all the objects.*
  - The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static member variables are stored separately rather than as a part of an object.
    - **Syntax defining a static data member:**

```
data_type class_name :: static_variable_name;          //assigned 0.
data_type class_name :: static_variable_name=value;    //for assigning some value
```

  - Static member variables are associated with the class rather than with any particular object, so they are also known as ***class variables***.

Example 3:

```
class test
{
        private:
                static int count;
                int m_nID;
        public:
                void Getdata(int a)        // function defined inside the class
                {
                        m_nID = a;
                        count++;
                }
        void GetCount(void)
                {
                        cout<<"Count is: " <<count <<endl;
                }
};
int test::count;          // definition of static data member, automatically assigned 0.

int main()
{
   test cFirst, cSecond, cThird;              // objects are created, count is initialized to 0.
cout<<" Before reading data: " << endl;
                cFirst.GetCount();
                cSecond.GetCount();                          // display count
                cThird.GetCount();
cout<<" Reading data..... " << endl;
        cFirst.Getdata(120);
        cSecond.Getdata(23); // getting data into objects
        cThird.Getdata(11);
cout<<" After reading data: "<< endl;
                cFirst.GetCount();
                cSecond.GetCount();                      // display count
                cThird.GetCount();
   return 0;
}
```

The static variable ***count*** is automatically initialized to 0 when objects are created. It is incremented whenever data is read into objects.

Initial value can be assigned to static variable as;

int test::count = 10; //10 is assigned to static data member count.

10.2 Static Member Functions

- Member functions that are declared using ***static*** keyword are called **static member functions** of the class.
- A **static member function** can have access to only **static class members** (both function and variables) declared in the same class.
- A **static member function** can be called using the class name (instead of its objects) as follows:
  - class name :: function_name(Args.);

Example 4:

```
#include<iostream>
using namespace std;
class test
{
private:
        static int count;
        int m_nID;
public:
        void Getdata(int a)        //function defined inside the class.
        {
                m_nID = a;
                count++;
         }

        static void GetCount(void)              // static function so can't access non-static class member.
        {
                 cout<<"Count is: " <<count <<endl;
                 cout<<" m_nID :" << m_nID;                    // illegal
        }
};
int test::count;                          // definition of static data member.
```

```
int main()
{
   test t1;                          //object created, so count is initialized to 0.
   t1.Getdata(120);
   test :: GetCount();               // display count, u can also use t1.GetCount().
   return 0;
}
```

11. Objects as Function Arguments

Like any other built-in type objects can be used as function arguments. This can be done in three ways:

- **Pass by value or call by value.**
- **Pass by reference or call by reference.**
- **Pass by address(pointer) or call by address**

If we pass object value to function that is called by value, in this case any changes made to the object inside the function do not affect the actual objects.

In call by reference, objects are passed through reference. If we pass objects as a call by address, we pass the address of the object. Therefore, any change made to the object inside the function will reflect in the actual object. The object which is used as an argument when we call the function is known as actual object and the object which is used as an argument within the function definition is called formal object.

11.1 Pass by value:

- A copy of entire object is passed to the function. Any changes made to the object inside the function do not affect the object used in the function call.
- Consider the following program: in this program function ***sum ()***, takes two objects as arguments.

Example 5

```
#include<iostream>
using namespace std;
class time
{
        int hrs;
        int minutes;
   public:
        void getdata(int a ,int b)
        {
                hrs = a;
```

```
                minutes =b;
        }
        void display(void);
        void sum(time, time);          //function declaration with objects as function arguments
};
void time :: sum(time t1, time t2)          // function definition outside the class
   {
      minutes = t1.minutes+t2.minutes;      // inside the function body, minutes and hrs belongs to calling
                                            object, in this case T3(look function call in main()).
      hrs = minutes/60;
      minutes =minutes%60;
      hrs=hrs+t1.hrs+t2.hrs;
   }
void time:: display(void)
{
   cout<< hrs<<"  hours and  " << minutes<<"  minutes";
}
int main()
{
        time T1,T2,T3;                  // three objects are created.
        int a,b,c,d;
   cout<<"enter values for a,b,c,d: "<<endl;
   cin>>a>>b>>c>>d;
        T1.getdata(a,b);
         T2.getdata(c,d);
   cout<<"T1 :  " <<"\t";
        T1.display();
   cout<<endl;
   cout<<"T2 : " <<"\t";
        T2.display();
   cout<<endl;
        T3.sum(T1,T2);                  // function call with object as arguments.
   cout<<"T3 : " <<"\t";
        T3.display();
   return 0;
}
```

11.2 Pass by Reference:

- A reference of an object is passed to the function. Formal objects now become the aliases to actual objects. So any changes made to the object inside the function will automatically reflect in the actual object.
- Consider the following program: in this program function ***sum ()***, takes reference of two objects as arguments.

Example 6:

```
#include<iostream>
using namespace std;
class time
{
   int hrs;
   int minutes;
   public:
   void getdata(int a , int b)
   {
      hrs = a;
      minutes =b;
   }
   void display(void);
   void sum(time &, time&); //function declaration, reference to objects(same as reference variable).
};

void time :: sum(time & t1, time & t2)          //function call, with reference to objects.
   {
      minutes = t1.minutes+t2.minutes;
      hrs = minutes/60;
      minutes =minutes%60;
      hrs=hrs+t1.hrs+t2.hrs;
   }
void time:: display(void)
{
   cout<< hrs<<" hours and " << minutes<< " minutes";
}
```

```
int main()
{
        time T1,T2,T3;
   int a,b,c,d;
   cout<<"enter values for a,b,c,d: "<<endl;
   cin>>a>>b>>c>>d;
        T1.getdata(a, b);
        T2.getdata(c, d);
   cout<<"T1 :  ";
        T1.display();
   cout<<"T2 : ";
        T2.display();
        T3.sum(T1, T2);
   cout<<"T3 : ";
        T3.display();
   return 0;}
```

11.3 Pass by Address or Call by address:

- If we pass objects as a call by address, we pass the address of the object. Therefore, any change made to the formal object inside the function will reflect in the actual object.

Example:

```
class time
{
        int hrs;
        int minutes;
  public:
        void getdata(int a ,int b)
        {
           hrs = a;
                minutes =b;
        }
        void display(void);
        void sum(time *, time*);        //function declaration with object pointers as function arguments.
};
```

- Like any other built-in type we can define object pointers. Once we define object pointer, we need to use (->) arrow operator to access the class members i.e., data members and member functions.
- Example: If time is a class;
  - time p;
    time * t = & p;
    t->getdata();
    t->display();

```
void time :: sum(time *t1, time *t2)          // function definition outside the class
   {
      minutes = t1->minutes+t2->minutes;      // inside the function body, minutes and hrs belongs to calling
      hrs = minutes/60;                       object, in this case T3(look function call in main).
      minutes =minutes%60;
      hrs=hrs+t1->hrs+t2->hrs;
   }
void time:: display(void)
{
   cout<< hrs<<"  hours and  " << minutes<<"  minutes";
}
int main()
{
         time T1, T2, T3;                 // three objects are created.
         int a,b,c,d;
   cout<<"enter values for a,b,c,d: "<<endl;
   cin>>a>>b>>c>>d;
         T1.getdata(a,b);
          T2.getdata(c,d);
   cout<<"T1 :  " <<"\t";
         T1.display();
   cout<<endl;
   cout<<"T2 : " <<"\t";
         T2.display();
   cout<<endl;
         T3.sum(&T1, &T2);              // function call with object address as arguments.
   cout<<"T3 : " <<"\t";
         T3.display();
   return 0;
}
```

12. Returning Objects

Like built-in type *int, float, double* etc., a function can return object.

Example 7:

```
#include<iostream>
using namespace std;
class Complex
{
      float real;
      float img;
   public:

void getdata(float a , float b)
   {
      real = a;
      img =b;
   }
   void display(void);
    Complex Difference(Complex);   //Remember, here function Difference has return type Complex.
};
Complex Complex::Difference(Complex C)
   {
      Complex s;
      s.real = real - C.real;
      s.img = img - C.img;
      return s;
   }
void Complex:: display(void)
{
   cout<<real<<" + " <<img<<"i"<<endl;
}
int main( )
{
   Complex C1, C2, C3;
   float a,b,c,d;
   cout<<"enter values for a, b, c, d: "<<endl;
   cin>>a>>b>>c>>d;
```

```
    C1.getdata(a,b);
    C2.getdata(c,d);
    cout<<"C1 : ";
       C1.display();
    cout<<"C2 : ";
       C2.display();
    C3 = C1.Difference(C2);          //This call to Difference() function returns an object, and
    cout<<"C3 : ";                   //content of that object is assigned to C3.
        C3.display();
    return 0;
}
```

Exercise:

1. Create a class called Dollar with two data members' dol and cent. Construct different member functions with the following operations.
   - To input data for Dollar objects.
   - To show the data of Dollar objects.
   - Member function to add two Dollar objects and then return the result.
2. WAP to multiply two matrix objects. Your class should include a multiplication() function that receives two matrix objects as arguments and returns a new matrix object containing their multiplication result.

13. Friend Functions

The concepts of data hiding and encapsulation dictate that private members of a class cannot be accessed from outside the class, that is, non-member functions of a class cannot access the non-public members (data members and member functions) of a class. However, we can achieve this by using friend functions.

A *friend function* is a function that is not a member function of a class but has full access right to private data members. Friend functions are declared with the 'friend' specifier in a class.

Friend functions of a Class possesses following special characteristics:

- A friend function is not in the scope of the class in which it has been declared as friend.
- It cannot be called using the object of that class.
- It can be invoked like a normal function without any object i.e., calling object does not needed.
- Unlike member functions, it cannot use the member names directly.
- It can be declared in public or private part without affecting its meaning.
- A function may be declared as friend to more than one class.
- A friend function may or may not be a member of another class.

- Usually, it has the objects as arguments.
- The declaration is preceded by the keyword “friend”. Its definition is written somewhere outside the class. To make an outside function “friendly” to a class, we have to declare this function as a friend of the class as shown below:

```
class test
{
        …….            // private members
        …….
        public:
        ……
        ……..
        friend void sum (arguments)    // generally objects are arguments for friend functions.
};
```

- **To define a friend function:**

```
void xyz(formal args.)      //Remember, friend function definition does not contain friend keyword.
{
        //function body
}
```

  - It should be noted that a friend function definition does not use friend keyword or scope resolution operator (: :).
- **To call a friend function**

```
xyz(actual args.)           //does not needs a calling object.
```

Example 8:

```
class sample
{
        int a, b;
    public:
        void setvalue()
        {
            a = 25;
            b = 40;
        }
        friend float mean(sample s);
};
float mean(sample s)
{
    return float(s.a + s.b)/2;
}
```

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

Output:

Mean value = 32.5

Another Example:

```
#include<iostream>
using namespace std;
class sample
{
     int x;
     int y;
  public:
    void getdata(int, int);
    friend int add(sample);

};
void sample ::getdata(int a, int b)
{
  x=a;
  y=b;
}
 int add(sample s)
{
  return s.x+s.y;
}
```

```
int main()
{
  sample s;
  int d=7;
  int c =4;
  s.getdata(c, d);
  cout<<"Sum is:"<<add(s); //friend function call, same as normal function.
  return 0;
}
```

13.1 Friend as a bridge:

Furthermore, a friend function also acts as a bridge between two classes. If there is a situation that requires a function to operate on objects of two different classes, it can be accomplished by declaring the same function as a friend in both the classes, taking the object of the two classes as arguments so that the function can operate on their private data.

Example 9:

```
class beta;                    // forward declaration
class alpha
{
        private:
                int data;
        public:
                void setdata(int d)
                {
                        data = d;
                }
                friend int frifunc(alpha, beta);          //friend function declaration
};
class beta
{
        private:
                int data;
        public:
                void setdata(int d)
                {
                        data = d;
                }
        friend int frifunc(alpha, beta);                  //friend function declaration
};
int frifunc(alpha a, beta b)
{
        return a.data + b.data;
}
```

```
int main()
{
        alpha aa;
        aa.setdata(7);
        beta bb;
        bb.setdata(3);
        cout<<frifunc(aa, bb);
        return 0;
}
```

Output:

10

***Another Example*:** This program swaps private members of two classes. Here objects are passed by reference.

```
class beta;       //forward declaration
class alpha
{
   int value1;
   public:
   void getdata(int a)
  {
      value1 = a;
  }
    void display(void)
   {
      cout<<" Value1:  " << value1;
   }
 friend void swap(alpha &  , beta & );                    //friend function declaration.
};
class beta
{
  int value2;
  public:
   void getdata(int b)
   {
      value2 = b;
   }
```

```
    void display(void)
    {
        cout<<" Value2:  " << value2;
    }
   friend void swap(alpha &  , beta &  );         //friend function declaration.

};
 void swap(alpha  &s1, beta & t1)      //friend function definition.
{
    int temp;
    temp = s1.value1;
    s1.value1 = t1.value2;
    t1.value2 = temp;
}
int main()
{
    alpha s;
    beta t;
    s.getdata(12);
    t.getdata(45);
    cout<<" values before swapping: " <<endl;
    s.display();
    cout<<endl;
    t.display();
    swap(s, t);               //calling friend function.
    cout<<endl;
    cout<<" Values after swapping: " <<endl;
    s.display();
    cout<<endl;
    t.display();
    return 0;
}
```

13.2 Member functions of one class can be friend functions of another class:

Member functions of one class can be friend functions of another class, as shown below:

```
class X
{
                .....
                .....
        public:
                int fun1();
                ..........
                ..........
};
class Y
{
                .........
                .........
        public:
                .......
                friend int X: : fun1();  // fun1() is a member function of class X,  now also is a friend of class Y.
};
```

Example 10:

```
#include <iostream>
using namespace std;
class beta;
class alpha
{
        public:
        void swap(beta &);                      // member function of class alpha.
};
class beta
{
   private:
                int x;
                int y;
```

```
        public:
            void setdata(int d, int r)
                {
                    x = d;
                    y = r;
                }
                void display()
                {
                    cout<<"X: " <<x<<" y: "<<y<<endl;
                }
                friend void alpha::swap(beta &);

};
void alpha::swap(beta & A)
{
   int temp;
   temp=A.x;
   A.x=A.y;
   A.y=temp;}
int  main()
{
        alpha a;
        beta b;
        b.setdata(99,100);
        b.display();
   a.swap(b);
   b.display();
   return 0;
}
```

Another Example:

Can u guess what will be the output of the following program?

```
#include <iostream>
using namespace std;
class beta;                    //forward declaration
class alpha
{
   string name;
        public:
        void getdata(beta&); //member function of class alpha, taking object of class beta as an argument.
        void display(beta);    //member function of class alpha, taking object of class beta as an argument.
};
class beta
{
   private:
                int x;
                int y;
        public:
           void setdata(int d, int r)
                {
                        x = d;
                        y = r;
                }
                friend void alpha::display(beta);       //member of class alpha declared friend in class beta.
                friend void alpha::getdata(beta &);    //member of class alpha declared friend in class beta.
};
```

```
void alpha::display(beta D)
{
   cout<<"You entered the string: "<<name<<endl;
   cout<<"X: "<<D.x<<endl;
   cout<<"Y: "<<D.y<<endl;
}
void alpha::getdata(beta &B)
{
        cout<<"Enter name: "<<endl;
        cin>>name;
        B.setdata(4, 6);
}
```

```
int  main()
{
        alpha a;
        beta b;
        a.getdata(b);
        a.display(b);
        return 0;
}
```

14. Friend Class

Classes that are declared with the ***friend*** keyword in another class are called ***friend classes*** of that class. Friend classes are used when we have to make all member functions of one class friend of another class. In this case, member functions friend class can access private data member of another class.

```
class X
{       …….
        public:
        void fun1();
        void fun2();
        …….
};
class Y
{       ……
        friend class X;          // all member functions of class X,  are now friend to class Y.
};
```

Example 11:

```
#include <iostream>
using namespace std;
class beta;
class alpha
{
        private:
```

```
                int x;
        public:
          void setdata(int d)
             {
                x = d;
              }
  friend class beta;  //Now, class beta is friend class to alpha, all member functions of class beta now
                      //become friend to class alpha.
};
class beta
{
        int p;
 public:
        void setdata(int d)
           {
                p = d;
            }
        void diff(alpha a)
                {
                        cout<<"Difference is: "<<p-a.x<<endl;
                }
        void add(alpha c)
                {
                        cout<<" sum is : "<<p+ c.x;
                }
};
int  main()
{
        alpha a;
        beta b;
        a.setdata(99);
        b.setdata(100);
        b.diff(a);
        b.add(a);
        return 0;
}
```

In the above program, in class **alpha** the entire class **beta** is declared as **friend**. Hence, all the member functions of **beta** can access the private data of **alpha**.

Chapter 6

Constructors and Destructors

6.1 Constructors

A constructor is a special member function whose task is to initialize the objects of its class. It has the name same as that of class name. The constructor is automatically invoked whenever an object is created. It is used for automatic initialization of objects. *Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function.* It is called constructor because it constructs the values of data members of the class.

For example,

```
class rectangle
{
        private:
                int length;
                int breadth;
        public:
                rectangle( )                    //constructor with no argument
                {
                        length = breadth= 0;
                }
                rectangle(int x);               // constructor declared, constructor with one argument.

                ……
                ……
};
rectangle:: rectangle(int x)                    // constructor defined
{
        length = breadth= x;
}
```

Characteristics of constructor:

- ❖ They executed automatically when objects are created.
- ❖ Constructors have the same name as that of class name.
- ❖ They should be declared in the "public" section.
- ❖ They do not have return types, not even void and therefore, and they cannot return any values.
- ❖ Like other C++ functions, they can have default arguments.
- ❖ Constructors cannot be **virtual**.

- Constructors can be overloaded.

6.2 Types of Constructor

There are, basically, three types of constructor:

- Default constructor
- Parameterized Constructor
- Copy Constructor

6.2.1 Default Constructor

A constructor that does not take any parameter is called default constructor.

Calling default constructor:

rectangle rect1; ***(syntax:** class_name object_name;)*

There are three possible situations for this.

1. If we do not provide any constructor with a class, the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

```
class A
{
        //no constructor.
};
```

2. If we provide a constructor without any arguments then that is the default constructor.

```
class A
{
    A ()
     {}
    //or
   A (void)
     {}
};
```

3. If we provide constructor with all default arguments, then that can also be considered as the default constructor.

```
class A
{
    A (int x=5)
    {   }
};
```

6.2.2 Parameterized Constructor

A constructor that takes arguments is called a parameterized constructor. Arguments are passed when the objects are created.

Example:

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

Arguments are passed when the objects are created. This can be done in two ways.

-By calling the constructor explicitly

- **Example:** rectangle rect1 = rectangle(4, 4); ***//explicit call***

- By calling the constructor implicitly

- **Example:** rectangle rect1(4, 4); ***//implicit call***

6.2.3 Copy Constructor

In the C++ programming language, a copy constructor is a special constructor for creating a new object as a copy of an existing object.

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. A copy constructor is called when an object is created by copying an existing object. The process of initializing through a copy constructor is known as ***copy initialization***. A copy constructor takes a reference to an object of the same class as itself (as an argument). *We cannot pass the argument by value to a copy constructor.*

Example: The copy constructor may have the following signature (prototype):

```
        rectangle (rectangle &);          // reference to object of same class.
```

Calling copy constructor

rectangle obj2(obj1);

The above statement would define the object obj2 and initialize it to the values of obj1. The above statement can also be written as,

rectangle obj2 = obj1;

When no copy constructor is defined, the compiler supplies its own copy constructor.

Remember the statement

obj2 = obj1;

will not invoke the copy constructor. However, if obj1 and obj2 are objects, this statement is legal and simply assigns the values of obj1 to obj2, member by member. This is the task of the overloaded assignment operator (=).

COMPLETE EXAMPLE:

```
class alpha
 {
    int x, y;
    public:
      alpha (){};                 //default constructor
      alpha(int a, int b)         //parameterized constructor
      {
         x=a;
         y=b;
      }
      alpha (alpha & s)           //copy constructor
      {
         x =s.x;
         y=s.y;
      }
      void display(void)
      {
         cout<<"x is: " <<x<<" y is: " <<y<<endl;
      }
 };
```

```
int main()
{
    alpha a(4, 5);          //calling parameterized constructor; object a is created.
    alpha b(a);             //copy constructor called
    alpha c = b;            // copy constructor called again
    alpha d;                //d is created but not initialized.
    d = a;          //copy constructor not called, implicit overloading of assignment operator (=).
    a.display();
    b.display();
    c.display();
    d.display();
    return 0;
}
```

6.3 Constructor Overloading

We can define more than one constructor in a class which is called constructor overloading.

Example:

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

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

In the above example of class **Item**, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply two arguments, while the third one gets invoked when an object is passed as an argument.

6.4 Constructor with default argument

It is possible to define a constructor with default argument like in normal function. Constructor with default arguments are called ***default argument constructor***.

For example:

```
class complex
    {
        int x, y ;
    public:
        complex (int a=4, int b=0)      // constructor with two default arguments.
        {
            x=a;
            y=b;
        }
    } ;
```

In above example, the default value of b is zero i.e. (0 is assigned to y). We can create the following type of objects for above class:

1. complex C; //default value 4 is assigned to x and 0 is assigned to y.

2. complex c1(5) //default value 0 is assigned to y.

3. complex c2(5, 6) //Here, default values are overridden.

- Constructor A:: A(int i =0) is called default argument constructor, where A is a class.
  - It is important to distinguish between the default constructor A::A() and the default argument constructor A::A (int x = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor.
  - When both these forms are used in a class, it causes ambiguity for a statement such as

    A a;
  - The ambiguity is whether to call A::A() or A::A(int x = 0).

6.5 Dynamic initialization of objects

Class objects can be initialized dynamically (i.e. at the run time). The user provides the values at the run time.

Advantage: various initialization formats can be provided using constructor overloading.

Example:

```
#include<iostream>
 using namespace std;
 class alpha
{
    float x, y;
    public:
      alpha (){}; //default constructor
      alpha(float a, float b) //constructor with two parameters.
      {
         x=a;
         y=b;
      }
void display(void)
 {
    cout<<"x is: " <<x<<" y is: " <<y<<endl;
  }
};
int main()
{
    float s, t;
    cout<<"Enter of values: "<<endl;
    cin>>s>>t;
    alpha A(s, t);
    A.display();
   return 0;
}
```

6.6 Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release memory and to perform other "cleanup" activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

Destructor will automatically be called by a compiler to clean up storage taken by objects. The objects are destroyed in the reverse order from their creation in the constructor.

Characteristics:

- Destructor is invoked automatically, when an object goes out of scope (i.e. exit from the program, or block or function).
- Like constructors, destructors do not have a return value.
- Destructor never takes any argument. Hence, we can use only one destructor in a class (*i.e., they cannot be overloaded*).
- Like constructors destructors are also defined in the public section.

Example:

```
class A
{
        int a;
 public:
    A( )  //constructor
        {
           cout<< "\n object created\n";
           a=0;
        }
  ~A( )  //destructor
        {
           cout<< "Object destroyed \n";
        }
};
int main( )
{
         A s, z;
         return 0;

}//Remember destructor is invoked automatically.
```

Objects are destroyed in the reverse order by the destructor:

```
#include<iostream>
using namespace std;
class A
{
        int a;
        static int count;
  public:
            A( )    //constructor
      {
        count++;
        cout<< count<<"object created\n";
        a=0;
      }
   ~A( )      //destructor
      {
        cout<<count<< "object destroyed \n";
        count--;
      }
};
int A:: count;    //automatically initialized to 0.
int main( )
{
   A p, q, r; //3 objects are created and initialized to 0.
   return 0;
}
```

Output:

Another Example:

```
#include<iostream>
using namespace std;
int count = 0;
class A
{
        int a;
        static int count;
 public:
            A( )    //constructor
      {
        count++;
        cout<< count<<"object created\n";
        a=0;
      }
   ~A( )      //destructor
      {
        cout<<count<< "object destroyed is: "<<count<<endl;
        count--;
      }
};
```

```
int main( )
{
   A p, q, r;  //3 objects are created and initialized to 0.
   {
        cout<< "\n \n Enter Block1\n" ;
        A s;
   }
   {   cout<< "\n \n Enter Block2\n" ;
      A t;
   }
cout<< "\n \n Re-enter main\n" ;
   return 0;
}
```

Run the following program, and analyze the output:

```
#include <iostream>
using namespace std;
class Item
{
                int code, price;

        public:
                Item()  //Default Constructor
                {
                   code= price =0;
                   cout<<"Object created is:  "<<this<<endl;   //this is an automatic pointer to the calling
                                                                //object.
                 }
```

```
        Item(int c, int p)                  //Parameterized Constructor
                {
                        code=c;
                        price=p;
                        cout<<"Object created is:  "<<this<<endl;   //this is an automatic pointer to the
                                                                    //calling object.
                }
                Item(Item &x)               //Copy Constructor
                {
                        code= x.code;
                        price= x.price;
                        cout<<"Object created is:  "<<this<<endl; //this is an automatic pointer to the
                                                                  //calling object.
                }
                void display()
                {
                        cout<<"Code: "<<code<<endl<<"Price: "<<price<<endl;
                }
                ~Item()
                {
                        cout<<"Object Destroyed is: "<<this<<endl;
                }
};  //class definition ends here.
int main()
{
        Item I1;                //calling default constructor
        Item I2(102,300);       //calling parameterized constructor
        Item I3(I2);            //calling copy constructor
        I1.display();
        {
           Item I5;
           I5.display();
        }//local functional block ends here. Object I5 is destroyed.
        I2.display();
        I3.display();
        return 0;
```

} **//main() block ends here, so object destroyed in the order I3,I2,I1.**

6.7 Dynamic Objects

As discussed earlier, C++ supports dynamic memory allocation/de-allocation. C++ allocates memory and initializes the member variables.

An object can be created at run time; such an object is called a dynamic object. The construction and destruction of the dynamic object is explicitly done by the programmer. The new and delete operators are used to allocate and de-allocate memory to such objects.

A dynamic object can be created using the new operator as follows:

```
class_name *ptr = new classname;
```

The new operator returns the address of the object created, and it is stored in the pointer ptr. The variable ptr is a pointer object of the same class. The member variable of the object can be accessed using the pointer and -> (arrow) operator. A dynamic object can be destroyed using the delete operator as follows:

```
delete ptr;
```

The delete operator destroys the object pointed by the pointer ptr. It also invokes the destructor of a class.

The following program explains the creation and destruction of dynamic objects:

```
#include<iostream>
using namespace std;
class data
{
   int x, y;
public:
   data()
   {
      cout<<"\n Constructor is called ";
      x=10;
      y=50;
   }
   ~data()
   {
      cout<<"\n Destructor";
   }
```

```
    void display()
    {
        cout<<"\n x="<<x;
        cout<<"\n y="<<y;
    }
};
int main()
{
    data *d = new data;          // dynamic object
    d->display();
    delete d;                    // deleting dynamic object, at this point destructor is automatically invoked.
    return 0;
}
```

OUTPUT

```
Constructor
x=10
y=50
Destructor
```

6.8 const (Constant) Object and const Member Functions

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

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

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

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

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

Exercise:

1. Define a class Rectangle with following specifications;
    – Data Members:
        • Length
        • Width
    – Constructors to assign values to objects.
    – Member functions:
        • Area() to find & print the area
        • Display() to display length and width
    – A destructor, to destroy objects.
2. Define a class Time with following specifications;
    – Data members:
        • Hrs
        • Min
    – Constructors to assign values
    – Member functions:
        • AddTime() to add two time objects
    –Friend function Display() to display values.
3. Create a class complex and write a program to add two complex numbers using friend function.
4. Explain the different types of constructors in C++ with suitable examples.
5. Differentiate between default constructor and default argument constructor with suitable program code.
6. Explain constructor overloading.
7. Define destructor with its properties. Why it is used in C++? Clarify your answer with suitable example.
8. Explain the concept of copy constructor with suitable example.
9. WAP to count the number of objects that are created.
10. Design a class called Complex to represent complex numbers. WAP to add two complex numbers. Use the concept of friend function. The function should return an object of type Complex representing the sum of two complex numbers.
11. Create a class Matrix. Then write a program to find transpose of a given matrix object, which is passed as a function argument. Use the concept of friend function. Your friend function should return a transposed matrix object.
12. Is it possible to define private constructor and destructor? Support your answer with suitable example.
13. “Destructor can’t be overloaded”, justify this statement with suitable example.
14. WAP that shows objects are destroyed in the reverse order of their creation.

Chapter 7

Operator Overloading & Type Conversions

7.1 Operator Overloading

Operator overloading is one of the crucial feature of C++ language. The concept by which we can give additional meaning to an operator of C++ language is known as operator overloading. In other words, operator overloading refers to giving the normal C++ operators (such as +, *, <=, += etc.) additional meanings when they are applied to user-defined data types. We can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

For example, + operator in C++ work only with basic type like *int* and *float* means c = a + b is calculated by compiler if a, b and c are basic types, suppose a, b and c are objects of user defined class, compiler give error. However, using operator overloading we can make this statement legal even if a, b and c are objects.
Actually, when we write statement c = a + b (and suppose a, b and c are objects of class), the compiler call a member function of class. If a, b and c are basic type then compiler calculates a + b and assigns that to c.

There are two types of operator overloading:

1. Unary operator overloading (++, --, - etc.)
2. Binary operator overloading (+, *, -, / etc.)

Operator Overloading Restrictions

1. The precedence of the operator cannot be changed.
2. The number of operands that an operator takes cannot be altered.
3. The overloaded operators must have at least one user-defined operand.
4. Only the existing operators can be overloaded. New operators cannot be created.
5. Operator functions cannot have default arguments.
6. We cannot overload following C++ operators:

   Class member access operators (**.***) Scope resolution operator (**::**)

   Size operator (**sizeof**) Conditional operator (**?:**).
7. We cannot use friend functions to overload certain operators. These operators are:

   = assignment operator ( ) function call operator

   [ ] subscripting operator -> class member access operator

7. 2 Defining Operator Overloading

Operator overloading is done with the help of a special function, called ***operator function***. Operator function can be defined either friend function or member function of the class. If operator function is a member function it can be defined either inside or outside of the class definition. The general form of an operator function is:

```
return-type operator op(arglist.)
        {
                //Function body.
        }
```

```
friend return-type operator op(arglist.);
//if operator function is a friend function
```

Where **return-type** is the type returned by the operation, **operator** is the keyword, and **op** is the operator (+, -, *, etc.) of C++ which is being overloaded as well as function name and **arglist** is argument passed to function.

For example, to add two objects of type distance each having data members feet of type int and inches of type float, we can overload + operator as follows:

```
distance operator +(distance d2)
        {
                //function body.
        }
```

Here, distance is a class name, operator is a keyword, and + is an operator. And we can call this operator function with the same syntax that is applied to its basic types as follows:

```
d3 = d1 + d2;  //d1 is the calling object.
```

Note: Operator overloading is done through operator function. The operator function must be either a *non-static member function* or *friend function* of a class. Most important difference between a member function and friend function is that a friend function will have only one argument for unary operators and two for binary operators. While a member function will have no arguments for unary operators and only one for binary operators. *The reason is object used to invoke the member function is passed implicitly and thus it is available for the member function. In other word, member function can always access the particular object for which they have been called.*

7.3 Rules for Overloading Operators

- The overloaded operator must have at least one operand that is of user-defined type.
- Unary operators overloaded by means of a member function take no explicit arguments and return no explicit values, but those overloaded by means of a friend function, take one reference argument (the object of relevant class).
- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
- When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
- Binary operators such as +, -, *, / must explicitly return a value.

7.4 Overloading Unary Operators (pre++, post ++, pre --, post--, unary-)

Unary operators are those operators that act on a single operand. ++, -- etc. are unary operators. Here are some examples that show how unary operators can be applied to objects.

Syntax For Calling operator function:

op **object_name; `**

E.g.: ++ S, --S, -S etc.

For post-increment and post-decrement:

object_name op**;**

E.g.: S++, S--etc.

Example1: Unary operator overloading for Unary Minus (-) operator

```
class Space
{
   int x;
   int y;
   int z;
public:
   Space (){x=y=z=0;}            //default constructor.
   Space (int a, int b, int c)//Parameterized constructor.
   {
      x=a;
      y=b;
      z=c;
   }
  void operator-() //defined inside the class.
   {
      x= -x;
      y= -y;
      z= -z;
   }
   void display()
      {
         cout<<"x= " <<x<<" ";
         cout<<"y= "<<y<<" ";
         cout<<"z= "<<z<<endl;
      }
};

int main()
{
   Space A(4,-5,-6);
   cout<<" A: ";
   A.display();
   -A;          //Calling operator function,
                //equivalent to A.operator-();
   cout<<"-A: ";
   A.display();
   return 0;
}
```

//Program to overload unary minus operator using friend function.

```
class Space
{
   int x;
   int y;
   int z;
public:
   Space(){x=y=z=0;}
   Space(int a, int b, int c)
   {
      x=a;
      y=b;
      z=c;
}
```

```
    friend void operator-(Space &);   //friend function, so reference argument.
    void display()
    {
        cout<<"x= " <<x<<" ";
        cout<<"y= "<<y<<" ";
        cout<<"z= "<<z<<endl;
    }
};
void operator-(Space &S)
    {
        S.x= -S.x;
        S.y= -S.y;
        S.z= -S.z;
    }

int main()
{
    Space A(4, -5, -6);
    cout<<" A: ";
    A.display ();
    -A;                    //equivalent to operator-(A).
    cout<<"-A: ";
    A.display ();
    return 0;
}
```

Example2: Overloading Unary Operator (pre++)

```
#include <iostream>
using namespace std;
class rectangle
{
        int length;
        int breadth;
public:
        rectangle(int l, int b)
        {
                length = l;
```

```
        breadth = b;
    }
    void operator ++()          //operator function defined inside the class.
    {
        ++length;
        ++breadth;
    }

void display()
    {
        cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
     }
};

int main()
{
    rectangle r1(5, 6);
    cout<<"Before increment: "<<endl;
    r1. display();
    ++r1;                   //calling operator function, same as; r1.operator ++();
    cout<<endl<<"After increment: "<<endl;
    r1. display();
    return 0;
}
```

In this example, we used prefix notation. We can also use postfix notation as follows:

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

Here int isn't really an argument, and it doesn't mean integer. It is simply a signal to the compiler to create the postfix version of the operator. We can call this operator function as follows:

```
r1++;
```

(TRY IT OUT!!!)

//Program to overload pre-increment operator using friend function.

```
class rectangle
{       int length;
        int breadth;
public:
        rectangle(int l, int b)
                {
                        length = l;
                        breadth = b;
                }
   friend void operator ++(rectangle &);        //friend operator function declaration.
   void display()
      {
        cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
      }
};
 void operator ++(rectangle & R)
      {
        ++R.length;
        ++R.breadth;
      }
int main()
{
        rectangle r1(5, 6);
        cout<<"Before increment: "<<endl;
        r1. display();
        ++r1;                                   //calling operator function, same as; operator ++(r1);
        cout<<endl<<"After increment: "<<endl;
        r1. display();
        return 0;
}
```

Example3: Overloading pre-decrement(--) operator

```
#include<iostream>
using namespace std;
class Distance
{
int feet;
float inch;
public:
   Distance (int f, float i);
   void operator--(void);        //operator function declaration.
   void display();
};
Distance :: Distance (int f, float i)
{
   feet = f ; inch = i;
}
void Distance :: display()
{
   cout<<" Distance "<<endl;
   cout<<"feet: "<<feet<<endl;
   cout<<"inch: "<<inch<<endl;
}
void Distance :: operator --(void)      //operator function defined outside the class.
{
   feet--;
   inch--;
}
int main()
{
   Distance d(4,5);
   d.display();
   --d;            //calling operator function.
   d.display();
   return 0;
}
```

Exercise: Redo above program using friend function.

Operator Return Values

The operator--() function can return a value. If we use a statement like this

r1 = --r2;

For this we have to define the -- operator to have a return type object of a class in the operator--() function. That is the compiler is being asked to return whatever value r2 has after being operated on by the -- operator, and assign this value to r1.

Example 4: overloading for post-fix increment operator

```
class rectangle
{       int length;
        int breadth;
public:
        rectangle(){length=breadth=0;}
rectangle(int l, int b)
      {
                        length = l;
                        breadth = b;
      }
   rectangle operator ++(int)  //operator function defined inside the class.
      {
        Rectangle temp;
        temp.length=length+1;
        temp.breadth=breadth+1;
        return temp;
      }
 void display()
      {
        cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
      }
};
int main()
{
        rectangle r1(5, 6),r2;
        cout<<"Before increment: "<<endl;
        r1.display();
```

```
        r2 = r1++;        //calling operator function, post-fix notation.
        cout<<endl<<"After increment: "<<endl;
        r2.display();
        return 0;
}
```

7.5 Overloading Binary Operators(+,-,,/,<,>,+= etc.)*

Binary operators are those that work on two operands. Examples are +,-,*, /, % for arithmetic operations, +=,-=,*= and /= for assignment operations and >, <, <=,>=, == and! = for comparison operations.

Overloading a binary operator is similar to overloading unary operator except that a binary operator requires an additional parameter.

For binary operators, overloaded operator function can be invoked by;

X op Y, where X, Y are objects. It would be interpreted as X.operator op(Y) in case of member function and operator op(X, Y) in case of friend function by the compiler.

Example 5: Program for binary operator overloading for +.

```
class time
{
        private:
                int hour;
                int min;
        public:
                 time( ){ }
        time(int a, int b)
        {
                hour =a;
                min=b;
        }

      time operator +(time t2)          //operator function
     {
        time temp;
        temp.hour= hour + t2.hour;
        temp.min=min + t2.min;
        temp.hour =temp.hour + temp.min / 60;
        temp.min=temp.min % 60;
        return temp;
```

```
    }
void display()
    {
      cout<<"("<<hour<<", "<<min<<")"<<endl;
    }
};

int main()
{
  time d1(5,78);
  time d2(7,55);
  time t;
  t=d1 + d2;  //function call, left-hand operand d1 invokes the operator function, d2 is passed as
              argumnt
  d2.display();
  d1.display();
  t.display();
  return 0;
}
```

NOTE: Remember left hand operand is always responsible for calling operator function. So in this example data members of d1 are accessed directly and the data members of d2 (that is passed as argument) are accessed using dot operator. Thus, both the objects are available for the function.

Overloading Binary Operators Using Friend Function

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

Example 6: Overloading binary operator + using friend function.

```
class time
{
      private:
            int hour;
            int min;
      public:
             time( ){ }
```

```
time(int a , int b)
                {
                        hour =a;
                        min=b;
                }
        void display()
                {
                        cout<<"("<<hour<<", "<<min<<")"<<endl;
                }
        friend time operator +(time t1, time t2);                //friend operator function
};
time operator +(time t, time t2)          //operator function definition
      {
         time temp;
         temp.min=t.min+t2.min;
         temp.hour= t.hour+t2.hour;
         temp.hour =temp.hour+(temp.min)/60;
         temp.min=(temp.min)%60;
         return temp;
      }
int main()
{
   time d1(5,78);
   time d2(7,55);
   d1.display();
   d2.display();
   time t;
   t=d1+d2;    //calling operator function.
   t.display();
   return 0;
}
```

----------Examples Programs----------

A program for overloading + to add two C-Style strings:

```
#include<iostream>
#include<cstring>
 using namespace std;
class String
 {
      private:
       char str[80];
      public:
       String()  { strcpy(str,"Rahul"); }
       String(char s[])  { strcpy(str, s); }
       void display()  { cout<<str<<endl; }
       String operator +(String );
};
  String String:: operator+(String ss)
   {
      String temp;
      if(strlen(str)+strlen(ss.str)<80)
     {
        strcpy(temp.str, str);
        strcat(temp.str, ss.str);
     }
      else
     {
        cout<<"string overflow"<<endl;
     }
      return temp;
   }
```

```
int main()
 {
    String s1("Happy "); //calling one argument constructor
    String s2("New year"); //calling argument constructor.
    String s3;
cout<<"Strings before overloading "<<endl;
   s1.display();
   s2.display();
   s3=s1+s2;
cout<<"After overloading "<<endl;
   s3.display();
   getch();
   return 0;
 }
```

//Overloading < operator:

```
#include<iostream>
using namespace std;
class time
{
        private:
                int hour;
                int min;
        public:
                 time( ){}
                time(int a , int b)
                {
                        hour =a;
                        min=b;
                }
      int operator <(time t2)
      {
         int m = hour*60 + min;
         int n= t2.hour*60+t2.min;
         if(m<n)
            return 1;
         else
            return 0;
      }
void display()
      {
         cout<<"("<<hour<<", "<<min<<")"<<endl;
      }
};
int main()
{
   time d1(7, 41);
   time d2(7, 44);
   cout<<" d1: ";d1.display();
   cout<<" d2: ";d2.display();
```

```
 if(d1< d2)      //At this point operator overloading routine is invoked…
   {
      cout<<" d1 is less than d2.";
   }
   else
   {
      cout<<" d2 is less than d1.";
   }
   return 0;
}
```

//Overloading += operator:

```
#include<iostream>
using namespace std;
class time
{
        private:
                int hour;
                int min;
        public:
                 time( ){   }

time(int a , int b)
        {
                hour =a;
                min=b;
        }
```

```
void operator +=( time t2)
{
   hour+=t2.hour;
   min+=t2.min;
   hour+=min/60;
   min=min%60;
}
```

OR

```
time operator +=( time t2)
{
   time temp;
   hour+=t2.hour;
   min+=t2.min;
   hour+=min/60;
   min=min%60;
   temp.hour=hour;
   temp.min =min;
   return temp;
}
```

```
void display()
{
   cout<<"("<<hour<<", "<<min<<")"<<endl;
}
};
int main()
{
   time d1(5,78);
   time d2(7,55);
   d1.display();
   d2.display();
   d1+=d2;              //operator function calling, d1 is calling object.
   d1.display();
   return 0;
}
```

//Above program can be done using friend operator function:

```
#include<iostream>
using namespace std;
class time
{
        private:
                int hour;
                int min;
        public:
                 time( ){ }
        time(int a , int b)
        {
                hour =a;
                min=b;
        }
friend void  operator +=( time & t1, time & t2);
void display()
      {
        cout<<"("<<hour<<", "<<min<<")"<<endl;
      }
```

```
};
void  operator +=( time & t1, time & t2)
{
   t1.hour+= t2.hour;
   t1.min+= t2.min;
   t1.hour+= t1.min/60;
   t1.min= t1.min%60;
}
int main()
{
   time d1(5,78);
   time d2(7,55);
   d1.display();
   d2.display();
  operator +=(d1,d2);
   d1.display();
   return 0;
}
```

//Overloading == operator:

```
class Distance
{
int feet;
float inch;
public:
Distance()
{}
Distance(int f, float i)
{
        feet = f;
        inch = i;
}
void display()
{
        cout<<feet<<inch;
}
int operator ==(Distance);
};
```

```
int Distance::operator==(Distance d)
{
if(feet == d.feet && inch == d.inch)
return 1;
else
return 0;
}

int main()
{
   Distance d1(4,89.3);
   Distance d2(4,89.2);
   if (d1==d2)
      cout<<"Both distances are of same length";
   else
      cout<<"Different";
   return 0;
}
```

//Overloading == operator to compare two C-style Strings:

```
#include<iostream>
#include<cstring>
using namespace std;
class String
{
        char Str[200];
public:
String()
{}
String(char b[])
{
   strcpy(Str, b);
}
void display()
{
cout<<"String "<<Str;
}
int operator ==(String);
};
```

```
int String::operator==(String d)
{
        if(strcmp(Str, d.Str)==0)
                return 1;
        else
                return 0;
}
int main()
{
   String d1("Ajay ");
   String d2("Pandey ");
   if (d1==d2)
      cout<<"Both Strings are same ";
   else
      cout<<"Different";
   return 0;
}
```

Exercise:

1. WAP to overload == operator using friend function.
2. WAP to overload unary minus (-) operator to invert sign of data members of a distance object.
3. Write a program to overload arithmetic assignment operators -= for two time objects.
4. Write a program to overload <= and >= operators.
5. Write a program to concatenate two strings using + operator.
6. Write a program to compare two C-style strings using = = operator.(use strcmp())
7. WAP to perform A = B + 100; here A, B are class objects.
8. WAP to perform Z = 120/Y; here Z, Y are class objects.

7.6 Type Conversion (Data Conversion)

Data conversion is the process of converting one data type to another data type. The type conversions are automatic as long as the data types involved are built-in types. If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routines by ourselves.

There are four types of data conversions;

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

Syntax: data conversion between incompatible types

target_type target_type_variable_name = source_type_variable_name;

Here target type may be user defined type or built-in type and also source type may be either user defined type or built-in type.

Example:

```
Time t;
int a=700;
t = a;          //here built-in type int is converted to user defined type Time.
```

7.6.1 Conversion from one basic type to another basic type

There are two types of conversion for built-in data type (int, float, char etc.).

Implicit data Conversion*:* Compiler is responsible for this type of data conversion. We do not need to write any conversion routine. Compiler automatically converts one basic type to another basic type.

For example:

```
int x;
float y=3.14 ;
x=y;
```

In above example, we are assigning the value of variable y to variable x. To achieve this, the compiler first converts y into integer and then assign it to x.

Explicit Conversion*:* In some cases automatic conversion may not work. Users have to write conversion routine explicitly. Explicit conversion is performed using type cast operator. **Three forms of type cast operator;**

- type_name (expression) **//c++ notation**
- (type_name) expression **//c notation**

❖ variable1 = static_cast<target_type> (variable2), where variable1 is target type variable and variable2 is source type variable.

For example:

```
int i, sum; float avg;
avg = sum/ (float)i;            //c notation
avg = sum/ float (i);           //c++ notation
avg = static_cast<float>(sum)
```

7.6.2 Conversion from basic type to class type(Basic type to User defined type)

The conversion from basic to class type can be done by using constructor. It is sometimes called *conversion constructor*. The constructor in this case takes single argument whose type is to be converted.

Syntax:

```
class_name(an argument of basic type)
{
    //conversion routine.
}
```

Example: Program to convert built-in type *int* to user-defined type *Time*.

```
#include<iostream>
using namespace std;
class Time
{
    int hour;
    int min;
  public:
    Time()          //default constructor.
     {}
     Time(int t)            //Conversion constructor.
     {
         hour = t/60;
         min = t%60;
     }
```

```
        void show( )
        {
                cout<<hour<< " Hour and "<<min<<" Minutes";
        }
};
int main()
{
        int a = 789;
        Time T1 = a;   // uses one-argument constructor to convert integer to time.
        T1.show();
        return 0;
}
```

Example: Program to convert built-in type *float* to user-defined type *Distance*

```
class Distance
{
   int feet;
   float inches;
public:
   Distance()                    //default constructor.
   {}
   Distance(float meters)        //conversion constructor.
   {
      float f= meters*3.28;      //since,1 meter = 3.28 feet.
      feet = int(f) ;
      inches = (f-feet)*12;
   }
  void show( )
  {
   cout<< " distance is: " <<endl ;
   cout<<feet<< " Feet and "<<inches<<" Inches";
  }
};
```

```
int main()
{
   float meter;
   cout<<"Enter a distance in meters: "<<endl;
   cin>>meter;
   Distance d1= meter;          //uses one argument constructor to convert float to Distance.
   d1.show();
   return 0;
}
```

7.6.3 Conversion from class type to basic type (User-Defined to Basic Type)

When class type data is converted into basic type data, it is called class to basic type conversion. The conversion constructor does not support this operation. This type of conversion is done through overloaded casting operator function, also called *conversion function*.

The general form of a conversion function is;

```
operator type_name()
{
        //function statements.
}
```

This function converts a class type data to type_name. For example, the **operator** double() converts a class object to type double, **operator** int() converts a class type object to type int, and so on. Since it is a class member function, it is invoked by the object; therefore values inside the function belong to the object that invoked the function.

Properties of Conversion Function:

- ❖ It must be a class member.
- ❖ It must not specify a return type.
- ❖ It must not have any arguments.

Example: Program to convert user-defined type i.e, *Distance* to basic type *float* i.e., meters.

```
class Distance
{
        private:
                int feet;
                float inches;
        public:
        Distance(int f, int i)
        {
                feet=f;
                inches=i;
        }
      operator float()          //conversion function, converts Distance type to a basic type float.
        {
                float ft=inches/12 ;
                ft=ft+feet ;
                return(ft/3.28) ;
        }
```

```
};
int main()
{
	Distance d(12, 6.56);
	float x = d;		//conversion function calling, d is responsible for calling.
	cout<<"x = "<<x<<" meters.";
	return 0;
}
```

Example: Program to convert user-defined type i.e., *Time* to basic type *int* i.e, seconds.

```
#include<iostream>
using namespace std;
class Time
{
	private:
		int hour;
		int min;
	public:
		Time(int h, int m)
		{
			hour=h;
			min=m;
		}
		operator int()		//conversion function, converts class type Time to a basic type int.
		{
			int s = hour*60*60+min*60;
			return s;
		}
};
int main()
{
	Time T(10, 56);
	int s = T;		//conversion function calling, T is responsible for calling.
	cout<<s<<" seconds.";
	return 0;
}
```

7.6.4 Conversion from one class type to another class type (one user-defined to another user defined type)

When a data of one class type is converted into data of another class type, it is called conversion of one class to another class type.

We can convert one class (object) to another class type as follows:

object of X = object of Y, X and Y both are different type of classes.

Here X is an object of class X and Y is object of class Y. The class Y type data is converted to the class X type data and converted value is assigned to the X. The conversion takes place from class Y to Class X. Therefore, Y is source class and X is destination class. This type of conversion is carried out by either constructor or a conversion function. It depends upon where we want the routine to be located – in the source class or in the destination class.

a. Routine in the source class:

When the conversion routine is in source class, it is implemented as a conversion function i.e. the overloaded casting operator function.

Syntax:

```
operator target_type_name()  //here type-name is the destination class name.
    {
        //function body.
    }
```

Example: Program to convert one user defined type Rupee to another user defined type Dollar.

```
//Here, conversion routine is in the source class.
#include<iostream>
using namespace std;
class Dollar     //Destination class
{
   float Dol;
   float cent;
public:
   Dollar(float a, float c)
   {
      Dol=a;
      cent=c;
   }
```

```
    void display()
    {
     cout<<Dol<<" $ and "<<cent<<" Cents."<<endl;
    }
};
class Rupee     //Source class
{
    float Rs;
    float Paisa;
public:
    Rupee(float r, float p)
    {
        Rs=r;
        Paisa=p;
    }
    void display()
    {
        cout<<Rs<<"Rs. and "<<Paisa<<" Paisa."<<endl;

    }
    operator Dollar()     //this operator function is responsible for conversion.
    {
        float a, b;
        a=Rs/100;
        b=Paisa/100;
        return Dollar(a, b);
    }
};
int main()
{
    Rupee R(45.23,76.23);
    R.display();
    Dollar D = R;   //Here, R is a source type and D is a destination type.
    D.display();
    return 0;
}
```

b. Routine in the destination class:

When the conversion routine is in destination class, it is commonly implemented as a constructor. This constructor takes one argument of source type i.e. an object of source type.

Syntax:

```
class-name(an object of source type)
        {
            //conversion routine.
        }
```

Example: Program to convert one user defined type Rupee to another user defined type Dollar..

//Here, conversion routine is in the destination class.

```
#include<iostream>
using namespace std;
class Rupee
{
   float Rs;
   float Paisa;
public:
   Rupee(float r, float p)
   {
      Rs = r;
      Paisa = p;
   }
   void display()
   {
      cout<<Rs<<"Rs. and "<<Paisa<<" Paisa.";
   }
   float getRs()
   {
      return Rs;
   }
   float getPaisa()
   {
      return Paisa;
   }
};
```

```
class Dollar
{
   float Dol;
   float cent;
public:
   Dollar(){}
   Dollar(float a, float c)
   {
      Dol=a;
      cent=c;
   }
   Dollar(Rupee R)
   {
      Dol=R.getRs()/100;
      cent=R.getPaisa()/100;
   }
   void display()
   {
         cout<<Dol<<" $ and "<<cent<<" Cents."<<endl;
   }
};
int main()
{
   Rupee R(45.23,700.23);
   R.display();
   Dollar D = R;
   D.display();
   return 0;
}
```

Exercise:

1. WAP to convert Centigrade into Fahrenheit temperature. (F= 9*C/5 + 32)
2. Define two classes **Polar** and **Rectangle** to represents points in the polar and rectangle system, where;

   x = r cos a

   y = r sin a. Write a conversion routine to convert object of class **Polar** to object of class **Rectangle**.
3. Create classes called **Amount1**, **Amount2**, and **Amount3**. **Amount1** has data member Rs(float), Amount2 has data member Paisa(int) and Amount3 has data member Dollars(float). The classes must be defined in such a way that it is possible to convert **Amount2** into **Amount1** and **Amount1** into **Amount3**.
4. If operator function defined as member function then, write a program to achieve following call;

   A = B -2;

   Here, both A and B are objects of class DATA with one float type data member.
5. WAP to achieve following call, if operator function is defined as friend function;

   X = 4 + Y;

   Where, both X, Y are objects of type SAMPLE with one double type data member.

Chapter-8

Inheritance

8.1 Introduction

- Inheritance is the one of the most powerful feature of object-oriented programming.
- Inheritance or derivation is the process of creating a new class, called **derived class** from existing class, called **base class.**
- The derived class inherits some or all the properties from base class. The base class is unchanged by this.
- **A derived class has direct access to both its own members and the public, and protected members of the base class.**
- The idea of inheritance implements the "is a" relationship.

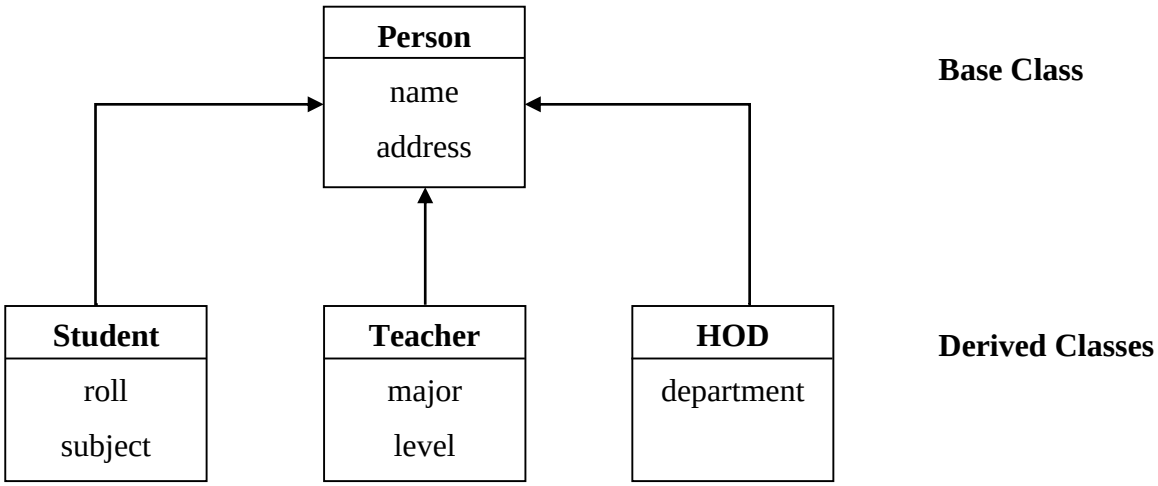


Fig: Inheritance

8.2 Benefits of inheritance

The benefits of inheritance are listed below:

- Inheritance provides the concept of reusability. This means additional feature can be added to an existing class without modifying the original class. This helps to save development time and reduce cost of maintenance.
- It also reduces coding effort. Code sharing can occur at several places.
- It will permit the construction of reusable software components. The new software system can be generated more quickly and conveniently.

8.3 Visibility Modifier (Access Specifier)

Three visibility labels: Private, protected, and public. **private** members of a class are accessible only from within the same class using member functions. private visibility label implements the concept of data hiding. We cannot access them outside of the class.

protected members are accessible by the member functions of the same class and also from any class immediately derived from it (i.e. Subclasses), and cannot be accessed by the functions outside these two classes. Thus, it forms the basis for inheritance.

Finally, **public** members are accessible from anywhere where the object is visible. It provide interface to interact with class members.

Visibility Modifier (Access Specifier)	Accessible from own class	Accessible from derived class	Accessible from objects outside class
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

8.4 Defining Derived Class (specifying Derived Class)/Defining inheritance

A derived class can be defined by specifying its relationship with the base class in addition to its own detail.

The general syntax is:

```
class derived_class_name : visibility_mode base_class_name
{
        //members of derived class.
} ;
```

Where, the colon (:) indicates that the derived_class_name is derived from the base_class_name. The visibility_mode is optional, if present, may be **private, protected or public**. The default visibility mode is private. Visibility mode controls the visibility and availability of inherited base class members in the derived class.

Four Important Points (regardless of access specifier):

1. The constructors and destructors of a base class are not inherited.
2. The friend functions and friend classes of the base class are not inherited.
3. The derived class does not have access to the base class's private members.
4. The derived class has access to all public and protected members of the base class.

Based on these three visibility labels, there are three forms of inheritance.

1. public Derivation or public inheritance
2. private Derivation or private inheritance
3. protected Derivation or protected inheritance

Public inheritance expresses an “is-a” relationship: a B is a particular type of an A, as a car is a type of vehicle, a manager is a type of employee, and a square is a type of shape.

Protected and private inheritance serve different purposes from public inheritance. Protected inheritance makes the public and protected members of the base class protected in the derived class. Private inheritance makes the public and protected members of the base class private in the derived class.

1. Public Derivation or public inheritance

```
class ABC : public XYZ        // public derivation
{
        //members of ABC.
} ;
```

When the base class is publicly inherited by derived class, the public and protected members are inherited. The public members of base class become public member in derived class. So, they can be accessed through the objects of the derived class. Whereas protected members of base class become protected in derived class. And the private members cannot be inherited.

Example:

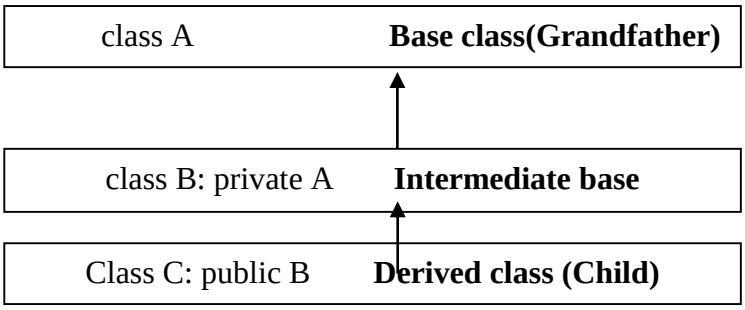
```
#include <iostream>
using namespace std;
class B
{
private:
        int x;
protected:
        int y;
public:
        int z;
void getdata( )
{
   cout<< "Enter 3 numbers= " ;
   cin>>x>>y>>z;
}
void showdata( )
{
        cout<< "x="<<x<<endl ;
        cout<< "y="<<y<<endl ;
        cout<< "z="<<z<<endl ;
}
} ;
```

```
class D : public B    //publicly derived class D from base class B.
{
private :
        int k;
public :
void getk( )
{
        cout<< "Enter k= ";
        cin>>k ;
}
void sum( )
{
        int s=y+z+k ;
        cout<< "y+z+k= "<<s<<endl ;
}

} ;
int main( )
{
        D d1;
        d1.getdata( ) ;          //why?
        d1.getk( ) ;
        d1.showdata( ) ;         //why?
        d1.sum( );
}
```

2. Private Derivation or private inheritance

```
class ABC : private XYZ       // private derivation
{
        //members of ABC.
} ;
```

When the base class is privately inherited, all the public and protected members of the base class become private members of the derived class. So, these cannot be accessed outside the class directly through the derived class object, but can be accessed by public functions in the derived class. Like general private members, these members can be used freely within the derived class. With a private inheritance, subsequent derived classes cannot inherit properties of base class. If class B is derived privately from class A, and class C is derived from class B, then class C cannot have access to any members of class A.




```
#include <iostream>
using namespace std;
class B
{
private :
   int x ;
protected :
   int y ;
public :
void getdata( )
{
   cout<< "enter x="; cin>>x ;
   cout<< "enter y=" ; cin>>y ;
}
void showdata( )
{
   cout<< "x="<<x<<endl ;
   cout<< "y="<<y<<endl ;
}
};
```

```
Class D: private B
                    //Private inheritance
{
   private :
      int z ;
public :
void setdata(int a, int b)
{
   z = a;
 }
void sum( )
{
   cout<< "Sum: "<<z+y ;
}

} ;
```

```
int main( )
{
   D d1;
   //d1.getdata();
   //d1.showdata();
   d1.setdata(4,6);
   d1.sum( );
}
```

3. Protected Derivation or protected inheritance

```
class ABC : protected XYZ    // protected derivation
{
        //members of ABC.
};
```

When base class is derived using protected mode, all the protected and public members of the base class become protected members of the derived class. This means, like a private inheritance, these members cannot be directly accessed through object of the derived class. But can be used freely within the derived class. Whereas, unlike a private inheritance, they can still be inherited and accessed by subsequent derived classes. In other words, protected inheritance does not end a hierarchy of classes, as private inheritance does.

Example:

```
#include <iostream>
using namespace std;
class B
{
private :
   int x ;
protected :
   int y ;
public :
void getdata( )
{
   cout<< "enter x="; cin>>x ;
   cout<< "enter y=" ; cin>>y ;
}
void display( )
{
   cout<< "x="<<x<<endl ;
   cout<< "y="<<y<<endl ;
}
};
```

```
class D : protected B    //Protected inheritance.
{
   private:
      int z ;
public:
void setdata(int a, int b )
{
   z = a;
   y = b;
}
void showdata( )
{
   cout<< "z: "<<z<<endl<<"y: "<<y ;
}
} ;
int main( )
{
   D d1 ;
   d1.getdata();     //won't work, it is protected member of D.
   d1.display();     //won't work, it is protected member of D.
   d1.setdata();
   d1.showdata( );
}
```

Summary:

Base Class Visibility	Derived Class Visibility		
	Public derivation	**Private derivation**	**Protected derivation**
private	Not inherited	Not inherited	Not inherited
protected	protected	private	protected
public	public	private	protected

8.5 Types of Inheritance

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

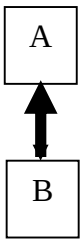
1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

8.5.1 Single inheritance

If a class is derived from only one base class, then that is called single inheritance. The figure below show this inheritance.

Example

```
class A                    //base class.
{
        members of A
};

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



Base Class

Derived Class

8.5.2 Multiple Inheritance

If a class is derived from more than one base class then inheritance is called as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes into one single class. It is like a child inheriting the physical features of one parent and the intelligence of another.

The syntax of multiple inheritance is:

class D: visibility B1, visibility B2 .....visibility Bn

{

member of class D.

} ;

The visibility can be private, public or protected. Note this is also possible that one visibility is public and another one is protected or private, etc.

For example:

(1) class D : public B1, public B2

{private : int a ;} ;

(2) class D : public B1, protected B2

{private : int a;} ;

(3) class D : private B1, protected B2, public B3

{private : int a ;} ;

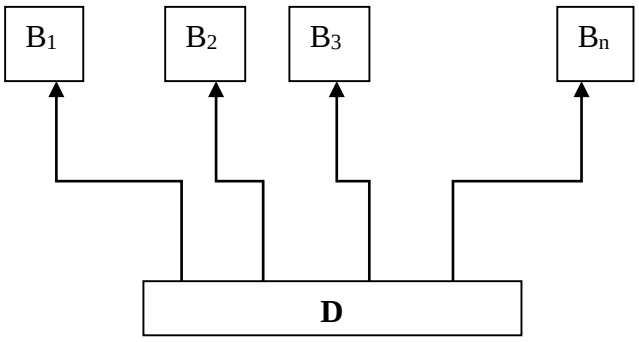


Fig: Multiple Inheritance

EXAMPLE:

```
#include<iostream>
using namespace std;
class biodata
{
  protected:
char name[20] ;
char semester[20] ;
int age ;
int rn ;
public:
void getbiodata( )
{
        cout<< "Enter name: "; cin>>name ;
        cout<< "Enter semester: "; cin>>semester ;
        cout<< "Enter age: " ; cin>>age ;
        cout<< "Enter rn: " ; cin>>rn ;
```

```
}
void showbiodata( )
{
        cout<< "Name:"<<name<<endl ;
        cout<< "Semester:"<<semester<<endl ;
        cout<< "Age:"<<age<<endl ;
        cout<< "Rn:"<<rn<<endl ;
}
} ;
class marks
{
  protected:
        char sub[10] ;
        float total ;
  public:
void getrm( )
{
        cout<< "Enter subject name:" ; cin>>sub ;
        cout<< "Enter marks:" ; cin>>total ;
}
void showm( )
{
        cout<< "Subject name:"<<sub<<endl ;
        cout<< "Marks are:"<<total<<endl ;
}
} ;
class final: public biodata, public marks
{
        char steacher[20] ;
public:
void gets( )
{
         cout<< "Enter your subject teacher:" ;
        cin>>steacher ;
}
void shows( )
```

```
{
        cout<< "Subject teacher:"<<steacher<<endl ;
}
} ;
int  main( )
{
        final f ;
        f.getbiodata( ) ;
        f.getrm( ) ;
        f.gets( ) ;
        f.showbiodata( ) ;
        f.showm( ) ;
        f.shows( ) ;
}
```

8.5.3 Hierarchical Inheritance

When two or more than two classes are derived from one base class, it is called hierarchical inheritance. With the help of hierarchical inheritance, we can distribute the property of one class into many classes. The diagram for hierarchical inheritance is given below.

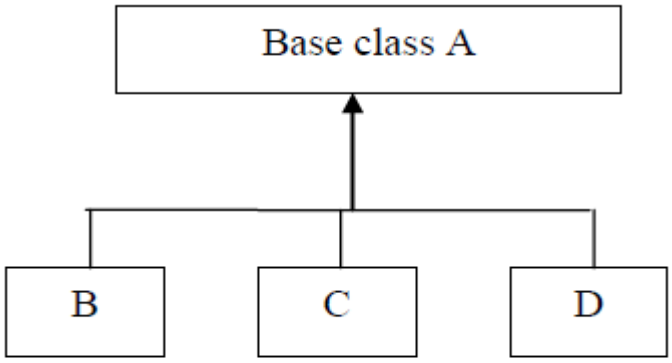


Fig: Hierarchical Inheritance

General Format:

```
        class B { ------ }
        class D1 : derivation B { ----- } ;
        class D2 : derivation B { ----- } ;
        class D3 : derivation B { ------} ; where derivation can be either public, protected or private type.
```

Example:

```
class A
        {
          …….
          …….
        };
class B : public A
        {
          ……
          ...……
        };
class C : public A
        {
          ……
          ……
        };
```

Here two classes B and C are derived from same base class A.

EXAMPLE:

```
#include<iostream>
using namespace std;
class B
{
   protected:
   int x, y ;
   public:
    void assign( )
      {
       x=10;
       y=20;
      }
}; //end of class B.
class D1: public B
{
 int s ;
   public:
```

```
    void add( )
    {
      s=x+y;
     cout<< "x+y: "<<s<<endl ;
    }
};//end of class D1.

class D2: public B
{
 int t ;
public:
        void sub( )
    {
         t=x-y ;
        cout<< "x-y: "<<t<<endl ;
    }
};//end of class D2.
class D3: public B
{
int m ;
  public:
    void mul( )
     {
        m=x*y;
        cout<< "x*y: " <<m<<endl ;
    }
};
```

```
int main()
{
        D1 d1 ;
        D2 d2 ;
        D3 d3 ;
        d1.assign( ) ;
        d1.add( ) ;
        d2.assign( ) ;
        d2.sub( ) ;
        d3.assign( ) ;
        d3.mul( ) ;
}
```

8.5.4 Multilevel Inheritance

The mechanism of deriving a class from another derived class is called multilevel inheritance. If A, B, and C are three classes. And if A is a base class, B is derived from A, again C is derived from B then derived class B is called intermediate base class. Since class B provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

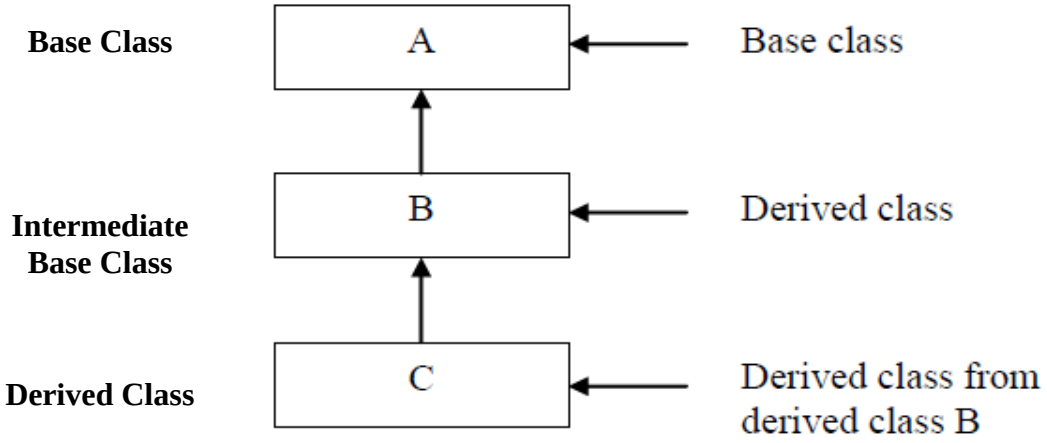


Fig: multilevel inheritance

For example:

```
class A {……..};                    //Base Class
class B : public A{……..};           //B derived from A
class C : public B{……..};           //C derived from B
```

In this example class B is derived from base class A and class C is derived from derived class B.

Complete Example:

```
class student
{
protected:
char name[20] ;
int rn ;
public :
void getdata( )
{
        cout<< "Student= "; cin>>name ;
        cout<< "Roll no.= "; cin>>rn ;
}
```

```
void showdata( )
{
        cout<< "Student= "<<name<<endl ;
        cout<< "Roll no="<<rn<<endl ;
}
};      // end of base class student.
class marks : public student
{
protected:
         int m1, m2 ;
public:
 void getm( )
   {
         cout<< "enter marks in Maths:" ;
         cin>>m1 ;
         cout<< "enter marks in English=" ; cin>>m2 ;
  }
void showm( )
{
        cout<< "Maths: "<<m1<<endl ;
        cout<< "English= "<<m2<<endl ;
}
}; //end of intermediate base class marks.
class result : public marks
{
        int total ;
public:
  void calculate( )
   {
     total=m1+m2 ;
   }
   void show( )
    {
        cout<< "Total marks= "<<total ;
     }
}; //end of derived class result.
```

```
int main( )
{
        result s1;
        s1.getdata();
        s1.getm();
        s1.calculate();
        s1.showdata();
        s1.showm();
        s1.show( );
        return 0;
}
```

In this example, class marks is derived from base class student and class result is derived from derived class marks.

8.5.5 Hybrid Inheritance

This inheritance is the combination of two or more types of inheritance.

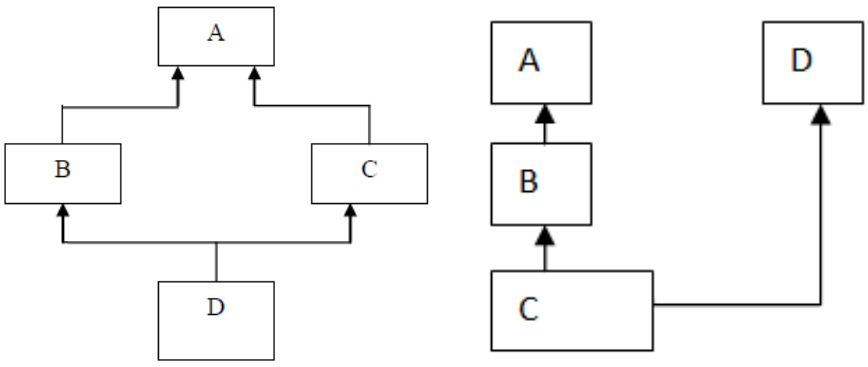


Fig: Hybrid Inheritance

```
#include <iostream>
using namespace std;
class B1
{
     protected :
 int x ;
    public:
   void assignx( )
   {
      x=2;
   }
```

```
}; //end of class B1.

class B2
{
   protected:
int k ;
  public:
  void assignk( )
    {
       k=80;
    }
};
class D1: public B1
{
   protected :
int y ;
  public :
   void assigny( )
    {
       y=40;
    }
 }; //end of class D1.

class D2 : public D1
{
   protected :
int z ;
   public :
    void assigz( )
    {
       z=60;
    }
} ;
class D3 : public B2, public D2
{
    private :
```

```
        int z;
int total ;
    public :
        void assignz(int p)
        {
           z=p;
        }
      void output( )
      {
           total=x+y+z+k ;
           cout<< "x+y+z+k=" <<total<<endl ;
      }
};
int main( )
{
          D3 s;
          s.assignx();
          s.assigny();
          s.assignz(8);
          s.assignk();
          s.output( );
}
```

8.6 Function Overriding

Defining a function in the derived class with same name as in the base class is called overriding. In this case, both base and derived class functions have same name, same number of arguments and similar type of arguments. Overriding is an object-oriented programming feature that enables a derived class to provide different implementation for a function that is already defined in its base class.

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.

Accessing the Overridden Function in Base Class From Derived Class:

If both derived class and base class contain same function name, then derived class object always accesses the derived class function. To access the overridden function of base class from derived class, scope resolution operator (::) is used.

Syntax:

base_class_name::function_name(); **//Calling function function_name() of base class base_class_name.**

Example:

```
class A
{
        public:
        void show()
                {
                        cout<<"This is class A";
                }
};


class B :  public A
{
        public:
        void show()     //overridden function.
                {
                        cout<<"This is class B"<<endl;
                }
};
int main()
{
  B b;
  b.show();             //invokes the member function from class B.
  b.A :: show();        //invokes the member function from class A.
}
```

8.7 Ambiguity in Multiple Inheritance

In multiple inheritance, when a member function with the same name appears in more than one base class, and if derived class does not have this member, and if we try to access this member using the objects of the derived class, it will be ambiguous. The ambiguity is that which base class function should be invoked by the compiler when we inherit those classes.

Example:

```
class A
{
        public:
          void display()
          {
            cout<< "This is base class A ";
          }
};
class B
{
    public:
          void display()
          {
            cout<< "This is base class B ";
          }
};
class C: public A, public B
{
        //here class C does not contain a function named display();
};
int main()
{
        C c;                //creating object of derived class C.
        c.display();    // ambiguous- will not compile, which function to call either of class A or class B.
}
```

Two ways to solve this problem

1. Using scope resolution operator.

To resolve this ambiguity we can call each of them using scope resolution operator.

```
int main()
```

```
{
        C c;                    //creating object of derived class C.
        c.A::display();         // now not ambiguous, calls display() function of class A.
        c.B::display();         // now not ambiguous, calls display() function of class B.
}
```

The problem is solved using the scope resolution operator to specify the class in which the function lies. Thus **c.A::display();** refers to the version of display() that's in the class A, while **c.B::display();** refers to the function in the class B.

2. Function Overriding

We can solve this problem by defining a same name function in the derived class. Simply we override those base class functions.

```
class A
{
      public:
        void display()
        {
          cout<< "This is base class A ";
        }
};


class B
{
   public:
        void display()
        {
          cout<< "This is base class B ";
        }
};
class C: public A, public B
{
        public:
          void display()        //function overriding, here class C contain a function named display().
            {
```

```
            A:: display();   //calls display() function of class A.
            B:: display();   //calls display() function of class B.
        }
};
int main()
{
    C c;            //creating object of derived class C.
    c.display();    // now not ambiguous, calls display() function of derived class C.
}
```

8.8 Ambiguity in Hybrid Inheritance and Virtual Base classes

Another kind of ambiguity arises if we derive a class from two classes that are each derived from the same base class. This creates a diamond shaped inheritance, shown in the figure. This ambiguity is due to several paths exist to a base class from the derived class. The ambiguity is that the derived class has multiple copies of the same base class i.e., duplicate sets of members inherited from a single base class, hence compiler can't decide which copy to use and signals an error.

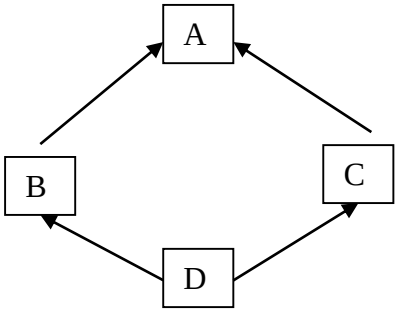


Fig: Multi-path Inheritance

Classes B and C are derived from same base class A, and class D is derived by multiple inheritance from both B and C. Ambiguity arises when we try to access a public members in class A from an object of class D. In this case both B and C contain a copy of that function, inherited from A. The compiler can't decide which copy to use, and signals an error.

Consider following example.

```
#include<iostream>
using namespace std;
class A
{
    protected:
        int adata;
    public:
```

```
        void getdata(int a)
                {
                        adata = a;
                }
        void display()
                {
                   cout<<" adtaa:"<<adata;
                }
};

class B :  public A
{
};
class C :  public  A
{
};
class D : public B, public C
{};

int main()
 {
        D d;
        d.getdata(4);    //ambiguous call to getdata() function, since two copies are available.
        d.display();     //ambiguous call to display() function.
}
```

To resolve this ambiguity we need to use the concept of virtual base classes.

Virtual Base Classes

The duplication of inherited members due to multiple paths can be avoided by making the common base class as virtual base class. This can be achieved by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class name with the word virtual.

When a class is made virtual, necessary care is taken by the compiler so that the duplication is avoided regardless of the number of paths that exist to the base class.

Example:

```
class A
{
        protected:
                int adata;
        public:
                void display()
                {
                   cout<<" adtaa:"<<adata;
                }
};
class B :   virtual public A      //parent1
{
};
class C : public virtual A        //parent2
{
};
class D : public B, public C
{
};
int main()
 {
        D d;
        d.getdata(4);    //unambiguous, since only one copy of is inherited.
        d.display();     //unambiguous, since only one copy of is inherited.
}
```

As we can see, the keyword virtual is used while deriving B, and C. Now that both B and C have inherited base class A as virtual, any multiple inheritance involving them will cause only one copy of base to be present. Therefore, in derived class D, there is only one copy of base A and hence the statements d.getdata(4), d.display() are perfectly valid and unambiguous.

Here, base class A is inherited as virtual base class to both B and C class, so derived class D has only one copy of the members of the base class A.

8.9 Constructor in derived classes

It is possible for the base class, the derived class or both to have constructor and / or destructor. There are several cases.

Case I: base class contains no constructor or contains only default constructor

If base class contains no constructor or contains only default constructor, the derived class does not need a constructor function.

Case II: If the base class contains constructor with arguments:

However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

Supplying values to base class constructor:

When applying inheritance we usually create objects using the derived class. If the base class contains a constructor it should be called from the initializer list in the derived class constructor as follows:

Syntax:

```
derived_class_constructor(arglist1, arglist2,…,arglistn):base1(arglist1), base2(arglist2), basen(arglistn)
{
        //body of derived constructor.
}
```

When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

Example:

```
class A
{
        protected:
                int adata;
        public:
                A(int a)
                {
                        adata = a;
                        cout<<"A is initialized"<<endl;
                }
};
```

```
class B :  public A
{
        int bdata;
        public:
                B(int x, int y) : A(x)
                {
                        Bdat a = y;
                        cout<<"B is initialized"<<endl;

                }
                void showdata()
                {
                        cout<<"adata = "<<adata<<endl <<"bdata = "<<bdata;
                }
};
int  main()
{
        B b(5, 6);
        b. showdata();
}
```

Case III: Case of multiple and multi-level inheritance:

In case of multiple inheritance, the base classes are constructed in order in which they appear in the declaration of the derived class. Similarly, in multilevel inheritance, the constructor will be executed in the order of inheritance.

Example1: For Multiple Inheritance

```
class A
{
        protected:
                int adata;
        public:
                A(int a)
                {
                        adata = a;
                        cout<<"A is initialized"<<endl;
                }
```

```
};
class B
{
        protected:
                int bdata;
        public:
                B(int b)
                {
                        bdata = b;
                        cout<<"B is initialized"<<endl;
                }
};
class C: public B,  public A    //constructor of base class B is executed first then of base class A.
{
        int cdata;
        public:
                C(int a, int b, int c) : A(a), B(b)
                {
                        cdata = c;
                        cout<<"C is initialized"<<endl;

                }

        void display()
        {
                cout<< "adata, bdata, cdata: " <<adata<<bdata<<cdata;
        }
};
int main()
{
        C v(4, 5, 6);
        v.display();
}
```

Example2: For Multilevel Inheritance

```
class base
{
   public:
   base( )
   {
      cout<< " Constructing base \n" ;
   }
};
class derived1: public base
{
public:
   derived1( )
   {
      cout<< " Constructing derived1 \n" ;
   }
};
```

```
class derived2 : public derived1
{
public:


   derived2( )
   {
      cout<< " Constructing derived2 \n"
;
   }
};
int main( )
{
   derived2 obj;
   return 0;
}
```

OUTPUT:

Constructing base

Constructing derived1

Constructing derived2

Another Example:

```
#include<iostream>

using namespace std;


class Base
{
   int i;
  public:
  Base ( int f)
 {
    i=f;
   cout << "Inside Base constructor" << endl;
  }
  ~Base ( )
  {
   cout << "Inside Base destructor" << endl;
  }
};
```

```
class Base1:public Base
{
   int y;
  public:
  Base1 ( int d, int f):Base(f)
  {
     y=d;
   cout << "Inside Base1 constructor" << endl;
  }
  ~Base1 ( )
  {
   cout << "Inside Base1 destructor" << endl;
  }
};

class Derived : public Base1
{
          int k;
  public:
  Derived  (int a, int b, int c ): Base1(b, a)
  {
     k=c;
   cout << "Inside Derived constructor" << endl;
  }
  ~Derived ( )
  {
   cout << "Inside Derived destructor" << endl;
  }
};

int main( )
{
  Derived x(4, 5, 8);
}
```

8.10 Execution order of constructors & destructors

The base class constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the definition of the derived class.

Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance. Furthermore, the constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared in the derived class.

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed.

Execution of base class constructors

Method of inheritance	***Order of execution***

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

The derived class destructor is executed first and then the destructor in the base class is executed. In case of multiple inheritance, the derived class destructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the destructors will be executed in the order of inheritance.

Example:

```
class base
{
   public:
   base()
   {
      cout<< "Constructing base \n" ;
   }
   ~ base()              //base class destructor
   {
      cout<< "Destructing base \n" ;
```

```
    }
};
class derived : public base
{
public:
    derived()
    {
        cout<< " Constructing derived \n" ;
    }
    ~ derived( )            //derived class destructor
    {
        cout<< "Destructing derived \n" ;
    }
};
int main( )
{
    derived obj;
    return 0;
}
```

Output:

Constructing base

Constructing derived

Destructing derived

Destructing base

8.11 Containership: (Aggregation)

A class can contain objects of another class as its members, which is called **containership** or **aggregation** or **nesting**. The class which contains the object is called container class. Containership implements a "has a" relationship. We say that a library has a book, meaning that each library includes an instance of a book. In object oriented programming, has a relationship occurs when object of one class is contained in another class. Containership is useful with classes that act like a data type. The object of these classes can be used almost like other variables in the class.

```
class A
 {
        ……
        ……
};
class B
{
        ……
        ……
};


class C          //Here, class C is a container class.
```

```
{
        …..
        A obj1;              //object of class A as a member.
        B obj2;              //object of class B as a member.
        …..
};
```

Example:

```
class Employee
{
        int eid, sal;
        public:
        void getdata()
        {
                cout<< "Enter id and salary of employee"<<endl;
                cin>>eid>>sal;
        }
        void display()
        {
                cout<< "Emp ID:"<<eid<<endl<<"Salary:"<<sal;
        }
};
class Company          //Company is a container class.
{
   int cid;
   char cname[45];
   Employee e;         //Company contains object of class Employee.
        public:
        void getdata()
        {
            cout<< "Enter id and name of the company:"<<endl;
            cin>>cid>>cname;
            e.getdata();
        }
        void display()
        {
            cout<<"Comp ID: "<<cid<<endl<<"Comp Name:"<<cname;
            e.display();
        }
};
```

```
int main()
{
        Company c;
        c.getdata();
        c.display();
}
```

Think: If Company contains 10 employees, what modification is needed in above program?

Solution:

```
class Employee
{
        int eid, sal;
      public:
      void getdata()
      {
            cout<< "Enter id and salary of employee"<<endl;
            cin>>eid>>sal;
      }
      void display()
      {
            cout<< "Emp ID:"<<eid<<endl<<"Salary:"<<sal;
      }
};
class Company
{
    int cid;
   char cname[45];
    Employee e[10];          //array of 10 Employees.
      public:
      void getdata()
      {
            cout<< "Enter id and name of the company:"<<endl;
            cin>>cid>>cname;
            for(int i=0;i<10;i++)
            {
               e[i].getdata();
            }
      }
```

```
void display()
        {
                cout<<"Comp ID: "<<cid<<endl<<"Comp Name:"<<cname;
                for(int i=0;i<10;i++)
               {
                   e[i].display();
               }
         }
};
int main()
{
        Company c;
        c.getdata();
        c.display();
}
```

Above Program Using Constructor

```
class A
{
   int age;
   char name[40 ];
   public:
   A(int a, char s[40])
   {
      age=a;
      strcpy(name, s);
   }
   void display()
   {
      cout<<" name: "<<name<<endl;
      cout<<" age: " <<age;
   }
};
```

```
class B
{
   float salary;
   A d;  //here, class B contains object of class A, as a member.
   public:
   B(int a, char c[ ], float r): d(a, c)    //provides data to class A.
   {
      salary= r;
   }
   void display()
   {
      cout<<" Salary: " <<salary<<endl;
      d.display();
   }
};
int main()
{
   B c(48,"Ram",87882.36);
   c.display();
}
```

Above program using standard C++ string

```
#include<iostream>
#include<string>
using namespace std;
class A
{
        int age;
        string name;
   public:
     A(int a, string s)
      {
        age=a;
        name=s;
     }
   void display()
      {
         cout<<" name: "<<name<<endl;
         cout<<" age: " <<age;
       }
};
class B
{
       float salary;
       A d;
   public:
       B(int a, string c, float r): d(a, c)
      {
          salary= r;
      }
      void display()
       {
          cout<<" Salary: " <<salary<<endl;
         d.display();
       }
};
```

```
int main()
{
   B c(48,"Ram",87882.36);
   c.display();
}
```

Exercise

1. Define a student class (with necessary constructors and member functions). Derive a Computer Science and Mathematics class from student class adding necessary attributes (at least three subjects). Use these classes in a main function and display the average marks of computer science and mathematics students.
2. Define a shape class (with necessary constructors and member functions). Derive triangle and rectangle classes from shape class adding necessary attributes. Use these classes in main function and display the area of triangle and rectangle.
3. Make a book class with member variables name, price, author_name and functions to get and show values of member variables. Make another class named grade with member variables level, 5 book objects, number_of_students, and get and show functions to get and show values of member variables. Also make main function to show uses of objects of class grade.

8.12 Local Classes

Classes can also be defined and used inside a function or a block. Such classes are called local classes.

Example:

```
void test (int a)
{
        ……
        ……
        class A
        {
                ……
                ……
        };
        ……
        ……
        A a;  //create object of type A.
        ……
}
```

Local classes can use global variables and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope resolution operator.