

PRACTICAL-1

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING IN C++

Introduction of Object Oriented Programming

Object Oriented Programming refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be grouped together. Most modern programming languages including Java, C/C++, and PHP, are object-oriented languages, and many older programming languages now have object-oriented versions.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added.

Basic Concepts of Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses “Objects and their interactions to design applications and computer programs.

There are different types of OOPs used, they are

Object:

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.

Class:

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction:

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class. This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

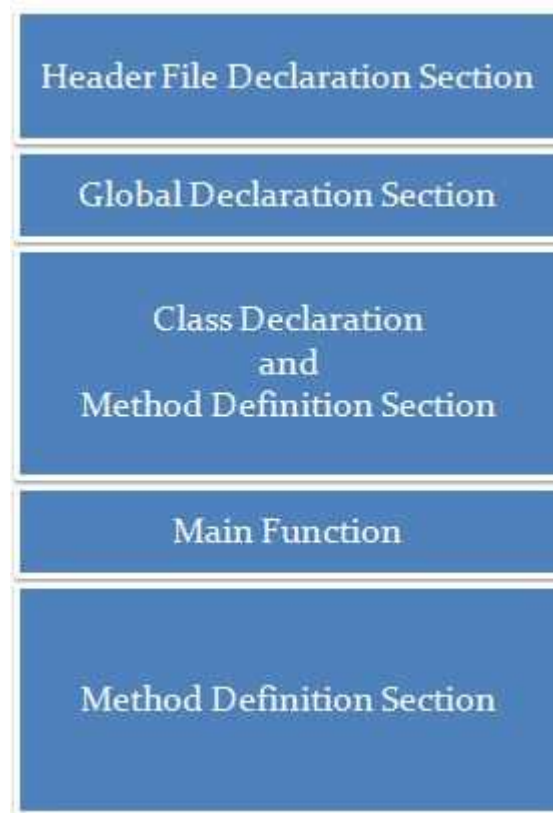
Overloading:

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Advantages of OOP:

- OOP provides a clear modular structure for programs.
- It is good for defining abstract data types.
- Implementation details are hidden from other modules and other modules has a clearly defined interface.
- It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- objects, methods, instance, message passing, inheritance are some important properties provided by these particular languages
- encapsulation, polymorphism, abstraction are also counts in these fundamentals of programming language.
- It implements real life scenario.
- In OOP, programmer not only defines data types but also deals with operations applied for data structures.

Basic Structure of C++ Programs



Structure of C++ Program

Basic Input/Output function in C++

The standard output stream (cout):

The predefined object `cout` is an instance of `ostream` class. The `cout` object is said to be "connected to" the standard output device, which usually is the display screen. The `cout` is used in conjunction with the stream insertion operator, which is written as `<<` which are two less than signs as shown in the following example.

The standard input stream (cin):

The predefined object `cin` is an instance of `istream` class. The `cin` object is said to be attached to the standard input device, which usually is the keyboard. The `cin` is used in conjunction with the stream extraction operator, which is written as `>>` which are two greater than signs as shown in the following example.

A Simple C++ Program

```
#include <iostream>

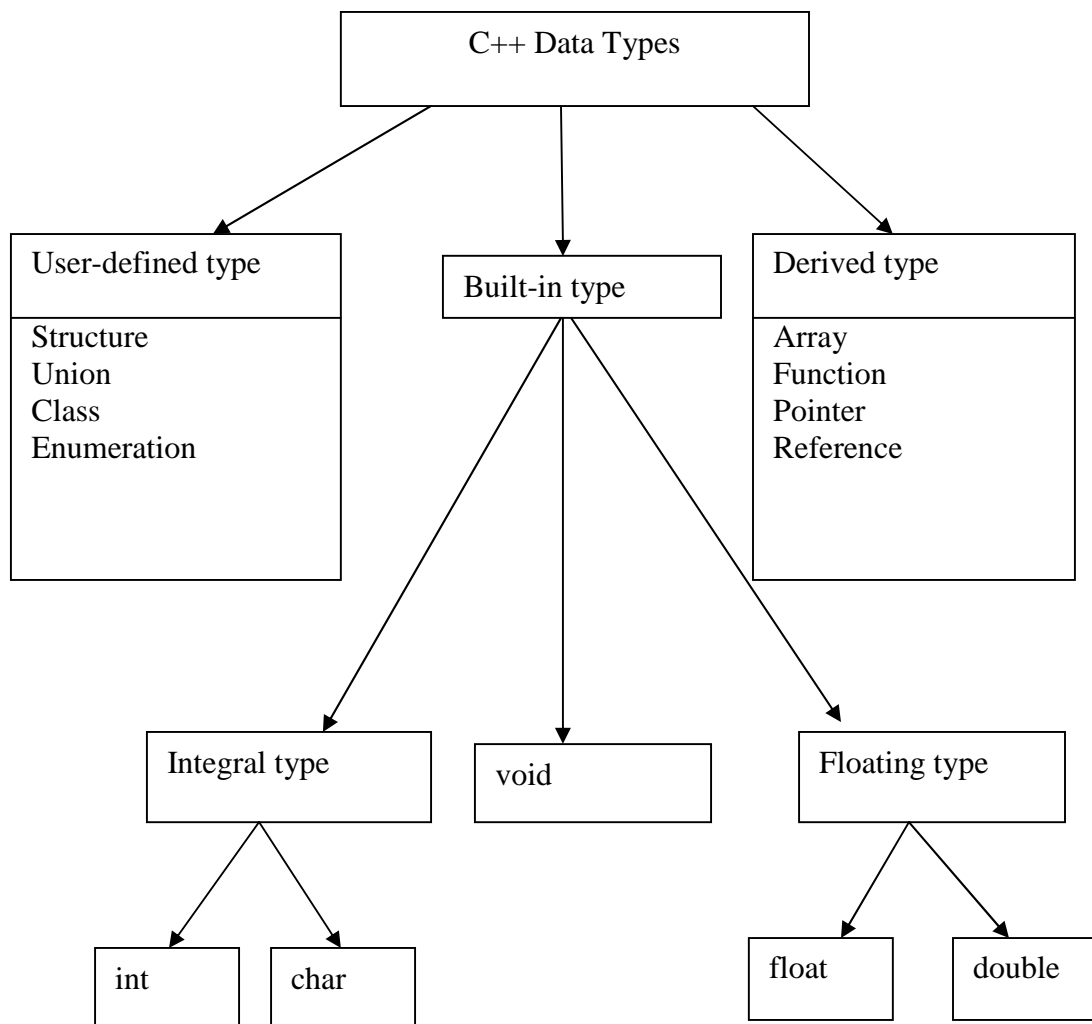
using namespace std;

int main( )
{
    char name[50];

    cout << "Please enter your name: ";
    cin >> name;
    cout << "Your name is: " << name << endl;

}
```

Basic Data Types



1)User-Defined Data Types

Structures

A structure is one or a group of variables considered as a (custom) data type. To create a structure, use the **struct** keyword, followed by a name for the object, at least followed by a semi-colon. Therefore, fundamentally, a structure can be created as:

```
struct SomeName;  
struct students  
{  
char name[10]; //name of student  
int SSN; // social security number  
char cardtype[10];  
float balance;  
}
```

Unions

A union is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

union [tag] { member-list } [declarators];

[union] tag declarators;

Parameters

tag

The type name given to the union.

member-list

List of the types of data the union can contain. See Remarks.

declarators

Declarator list specifying the names of the union.

Declaring a Union

Begin the declaration of a union with the union keyword, and enclose the member list in curly braces:

```
// declaring_a_union.cpp
union DATATYPE // Declare union type
{
    char ch;
    int i;
    long l;
    float f;
    double d;
} var1; // Optional declaration of union variable
```

Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language, this class describes the data properties alone. In C++ programming language, class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

Enumerated Data Type

Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values. Objects of these enumerated types can take any of these enumerators as value.

Their syntax is:

```
enum type_name {
    value1,
    value2,
    value3,
    .
    .
} object_names;
```

2) Derived Data Types

Pointers

A pointer is a memory variable that stores a memory address. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variable but it is always denoted by '*' operator.

```
int *x;  
float *f;  
char *y;
```

In the first statement 'x' is an integer pointer and it tells to the compiler that it holds the address of any integer variable. In the same way f is afloat pointer that stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.

Array

An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

```
Char string[3]="xyz";
```

Function

A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

Reference

A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made to any of them (say, a variable) are reflected in the other (on a reference).

Defining Constants

Symbolic constant is a quantity which does not change during the execution of the program. Constant qualifiers are keywords used to define a quantity as symbolic constant. steps and rules:

- 1) Const must be initialized.
- 2) Const is local to the function where it is declared.
- 3) By the qualifier extern along with const, it can be made global.
- 4) If data type is not given it is treated as integer

The qualifiers are

1) const.

```
const int KILL_BONUS = 5000;
```

2) enum.

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK }
```

Declaration of variables

A variable is a data name that may be used to store a data value. A variable may take different values at different times during execution.

```
type variable_list;
```

Some valid declarations are shown here

```
int i, j, k;  
char c, ch;  
float f, salary;  
double d;
```

Dynamic initialization of variables

C++, however, permits initialization of variables at run time. This is referred to as dynamic initialization.

```
int n = strlen(string);  
  
float area=3.14*rad*rad;
```

Reference variables

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

Basic Syntax

Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name, such as this "reference to an int

```
int& foo = ....;  
int x;  
int& foo = x;
```

```
// foo is now a reference to x so this sets x to 56
```



```
foo = 56;  
std::cout << x <<std::endl;
```

Functions taking References Parameters

Here's a simple example of setting up a function to take an argument "by reference", implementing the swap function:

```
void swap (int& first, int& second)  
{  
    int temp = first;  
    first = second;  
    second = temp;  
}
```

Both arguments are passed "by reference"--the caller of the function need not even be aware of it:

```
int a = 2;  
int b = 3;  
swap( a, b );
```

Scope Resolution Operator

Scope resolution operator (::) is used to define a function outside a class or when we want to use a global variable but also has a local variable with same name.

C++ programming code

```
#include <iostream>  
using namespace std;  
  
char c = 'a'; // global variable  
  
int main() {  
    char c = 'b'; //local variable  
  
    cout << "Local c: " << c << "\n";  
    cout << "Global c: " << ::c << "\n"; //using scope resolution operator  
  
    return 0;  
}
```

Memory Management Operators

There are two types of memory management operators in C++:

- new
- delete

These two memory management operators are used for allocating and freeing memory blocks in efficient and convenient ways.

New operator:

The new operator in C++ is used for dynamic storage allocation. This operator can be used to create object of any type.

General syntax of new operator in C++:

The general syntax of new operator in C++ is as follows:

```
pointer variable = new datatype;
```

For example:

```
int *a=new int;
```

The assignment can be made in either of the two ways:

1. int *a = new int;
2. *a = 20;

delete operator:

The delete operator in C++ is used for releasing memory space when the object is no longer needed. Once a new operator is used, it is efficient to use the corresponding delete operator for release of memory.

General syntax of delete operator in C++:

The general syntax of delete operator in C++ is as follows:

```
delete pointer variable;
```

Example:

```
1.      #include <iostream>
2.      using namespace std;
3.      void main()
4.      {
5.          //Allocates using new operator memory space in memory for
           storing a integer datatype
6.          int *a= new int;
7.          *a=100;
8.          cout << " The Output is:a= " << *a;
9.          //Memory Released using delete operator
10.         delete a;
11.
12.     }
```

Manipulators

Manipulator functions are special stream functions that change certain characteristics of the input and output. They change the format flags and values for a stream. The main advantage of using manipulator functions is that they facilitate that formatting of input and output streams.

endl

the endl is an output manipulator to generate a carriage return or line feed character

```
cout << " a = " << a << endl;
```

```
cout << " b = " << b << endl;
```

setw ()

The setw () stands for the set width. The setw () manipulator is used to specify the minimum number of character positions on the output field a variable will consume.

The general format of the setw manipulator function is

```
setw( int w )
```

Which changes the field width to w, but only for the next insertion. The default field width is 0.

For example,

```
cout << setw (1) << a << endl;  
cout << setw (10) << a << endl;
```

Program:

```
#include <iostream.h>  
#include <iomanip.h>  
void main (void)  
{  
    int a,b;  
    a = 200;  
    b = 300;  
    cout << setw (5) << a << setw (5) << b << endl;  
    cout << setw (6) << a << setw (6) << b << endl;  
    cout << setw (7) << a << setw (7) << b << endl;  
    cout << setw (8) << a << setw (8) << b << endl;  
}
```

Output of the above program

```
200    300  
200    300  
200    300  
200    300
```

Exercise:

- 1) **W.A.P to display name & address.**

2) W.A.P to perform all arithmetic operation on two integers scanned from user.

3) W.A.P to find avg. of three no. read from users.

4) W.A.P to convert temp from Fahrenheit to Celsius unit using eq: $C=(F-32)/1.8$

5) W.A.P to find no. of years, months, and days from entered no. of days using “%”.

6) W.A.P to swap any two no. using third variable.

PRACTICAL-2

Functions in C++ and Working with objects

Call by Reference and Return by Reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Example

```
#include <iostream>
using namespace std;
// function declaration
void swap(int &x, int &y);
int main ()
{
    // local variable declaration:
    int a = 100;
    int b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    /* calling a function to swap the values using variable reference.*/
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}
// function definition to swap the values.
void swap(int &x, int &y)
{
    int temp;
    temp = x; /* save the value at address x */
    x = y; /* put y into x */
    y = temp; /* put x into y */
    return;
}
```

Inline functions

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

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

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
    return (x > y)? x : y;
}

// Main function for the program
int main( )
{

    cout << "Max (20,10): " << Max(20,10) << endl;
    cout << "Max (0,200): " << Max(0,200) << endl;
    cout << "Max (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

Default Arguments

One of the most useful facilities available in C++ is the facility to defined default argument values for functions. In the function prototype declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default function prototype declaration. Fault arguments facilitate easy development and maintenance of program.

For example, The following program segment illustrates the default argument declaration:

```
#include <iostream.h>
void sum (int x = 10, int y = 20);
// function prototype declaration
void main () // with default argument list
{
    int a,b;
    sum (); // function calling
}

void sum (int a1, int a2) // function definition
{
    int temp;
    temp = a1+a2; // a1 = 10 and a2 = 20 by default arguments
}
```

Constant Arguments

The value of an argument that is declared const cannot be changed.

```
#include<iostream.h>
#include<conio.h>
//function prototype
float cir(float,float pi=3.142);
//main function
void main()
{
    //clear the screen.
    clrscr();
    //declare variable as float.
```

Prog.in C++ Laboratory Manual

```
float r,area;
//Input the radius.
cout<<"Enter the radius"<<endl;
cin>>r;
//calculate area using function.
area=cir(r);
//print area
cout<<"Area of circle is "<<area;
//get character
getch();
}
//function
float cir(float r,float pi)
{
float ar;
ar=pi*r*r;
return(ar);
}
Output
```

Enter the radius.

1

Area of circle is 3.142

Function Overloading

Function overloading means two or more functions can have the same name but either the number of arguments or the data type of arguments has to be different. Return type has no role because function will return a value when it is called and at compile time compiler will not be able to determine which function to call. using namespace std;

/ Function arguments are of different data type */*

```
long add(long, long);
float add(float, float);
```

```
int main()
{
    long a, b, x;
    float c, d, y;

    cout << "Enter two integers\n";
    cin >> a >> b;

    x = add(a, b);
```

```
cout << "Sum of integers: " << x << endl;

cout << "Enter two floating point numbers\n";
cin >> c >> d;

y = add(c, d);

cout << "Sum of floats: " << y << endl;

return 0;
}

long add(long x, long y)
{
    long sum;

    sum = x + y;

    return sum;
}

float add(float x, float y)
{
    float sum;

    sum = x + y;

    return sum;
}
```

Defining Class and Creating Objects

The mechanism that allows you to combine data and the function in a single unit is called a class. Once a class is defined, you can declare variables of that type. A class variable is called object or instance. In other words, a class would be the data type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

```
Class class_name
{
    Private:
    Variable declaration;
    Function declaration;
    Public:
    Variable declaration;
```

```
Function declaration;  
};
```

Object Declaration

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

```
Class_name obj1,obj2;
```

A Simple Program

```
#include<iostream>  
using namespace std;  
class programming  
{  
private:  
int variable;  
public:  
void input_value()  
{  
cout << "In function input_value, Enter an integer\n";  
cin >> variable;  
}  
void output_value()  
{  
cout << "Variable entered is ";  
cout << variable << "\n";  
}  
};  
main()  
{  
programming object;  
  
object.input_value();  
object.output_value();  
  
//object.variable; Will produce an error because variable is private  
  
return 0;  
}
```


Making an outside function Inline

```
Class item
{
    .....
    .....
    Public:
    Void getdata (int a, float b);
};
Inline void item :: getdata (int a, float b)
{
    Number = a;
    Cost=b;
}
```

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.

```
#include
using namespace std;
class set
{
    int m,n;
    public:
    void input(void);
    void display(void);
    int 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

Private Member functions

A private member function can only be called by another function that is member of its class. Even an object cannot involve a private function using the dot operator.

Class sample

```
{
    int m;
    void read (void);
public:
    void update (void);
    void write (void);
};
```

if s1 is an object of sample, then

```
s1.read(); // won't work; objects cannot access
           //private members
```

is illegal. However, the function read() can be called by the function update() to update the value of m.

```
void sample :: update (void)
{
    read(); // simple call; no object used
}
```

Array within a class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;
class array
{
    int a[size];
public:
    void setval(void);
    void display(void);
}
```

Static Data Member

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

```
#include <iostream>
using namespace std;
class alpha
{
private:
    int id;
    static int count;
public:
    alpha()
    {
        count++;
        id=count;
    }
    void print()
    {
        cout<<"My id is"<<id;
        cout<<"count is"<<count;
    }
};
int alpha ::count=0; //definition of count
void main ()
{
    alpha a1,a2,a3;
    a1.print();
    a2.print();
    a3.print();
}
```

}
Static Member Function

```
#include<iostream.h>
#include<conio.h>
class stat
{
    int code;
    static int count;
public:
    stat()
    {
        code=++count;
    }
    void showcode()
    {
        cout<<"\n\tObject number is :"<<code;
    }
    static void showcount()
    {
        cout<<"\n\tCount Objects :"<<count;
    }
};
int stat::count;
void main()
{
    clrscr();
    stat obj1,obj2;
    obj1.showcount();
    obj1.showcode();
    obj2.showcount();
    obj2.showcode();
    getch();
}
```

Output:

Count Objects: 2
Object Number is: 1
Count Objects: 2
Object Number is: 2

Array of Objects

Arrays of variables of type "**class**" is known as "**Array of objects**". The "identifier" used to refer the array of objects is an user defined data type.

```
#include<iostream>
#include<iomanip>
using namespace std;
class employee
{
    int no;
public:
    void getdata(void);
    void putdata(void);
};
void employee :: getdata(void)
{
    cout<<"enter no:";
    cin>>no;
}
void employee :: putdata(void)
{
    cout<<"No:"<<no<<endl;
}
int main()
{ int i;
  employee manager[4];
  for(i=0;i<3;i++)
  {
    manager[i].getdata();
  }
  for(i=0;i<3;i++)
  {
    manager[i].putdata();
  }
return 0;
}
```

Friend Function

A C++ friend functions are special functions which can access the private members of a class. A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

```
#include<iostream.h>
#include<conio.h>
class base
{
    int val1,val2;
public:
    void get()
    {
        cout<<"Enter two values:";
        cin>>val1>>val2;
    }
    friend float mean(base ob);
};
float mean(base ob)
{
    return float(ob.val1+ob.val2)/2;
}
void main()
{
    clrscr();
    base obj;
    obj.get();
    cout<<"\n Mean value is : "<<mean(obj);
    getch();
}
```

Output:

Enter two values: 10, 20

Mean Value is: 15

Exercises:-

1) Write a C++ Program to sort two numbers using call by reference

2) Write a C++ program to multiply two numbers using inline function.

3) Write a C++ program to find area of circle using formula $\pi * r^2$. take the value of pi as default argument in function.

4) Write a program to overload the max function

- (1) Find the max number from three numbers.**
- (2) Find the max string from three strings.**

5) Write a C++ program to enter the data of an employee and display the data of an employee using class and object.

6) Write a C++ program to enter number and cost of a class item using get data () and using put data () to display data. Write get data () outside the class item.

7) Create a 'DISTANCE' class with : - feet and inches as data members - member function to input distance

- member function to output distance

- member function to add two distance objects

Write a main function to create objects of DISTANCE class. Input two distances and output the sum.

8) Create a class 'COMPLEX' to hold a complex number. Write a friend function to add two complex numbers. Write a main function to add two COMPLEX objects.

9) Create a class called 'EMPLOYEE' that has - EMPCODE and EMPNAME as data members - member function getdata() to input data

- member function display() to output data

Write a main function to create EMP, an array of EMPLOYEE objects. Accept and display the details of at least 6 employees

10) Write a C++ program to show the use of static data member and static member function.

PRACTICAL-III **CONSTRUCTOR AND DESRUCTOR**

Constructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:

```
#include <iostream>

using namespace std;

class Line
{
    public:
        void setLength( double len );
        double getLength( void );
        Line(); // This is the constructor

    private:
        double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

// Main function for the program
int main( )
```

```
{
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
public:
    void setLength( double len );
    double getLength( void );
    Line(double len); // This is the constructor

private:
    double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
```

```
length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
    Line line(10.0);

    // get initially set length.
    cout << "Length of line : " << line.getLength() <<endl;
    // set line length again
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Multiple Constructors In a Class

A class can have multiple constructors that assign the fields in different ways. Sometimes it's beneficial to specify every aspect of an object's data by assigning parameters to the fields, but other times it might be appropriate to define only one or a few.

```
class integer
{
    int m,n;
    public:
    integer () { m=0,n=0 } //constructor 1
    integer ( int a, int b) { m=a; n=b; } // constructor 2
    integer (integer & i) // constructor 3
    { m=i.m ; n=i.n ; }
};
```

Constructor With Default Arguments

Like functions, it is also possible to declare constructors with default arguments. Consider the following example.

```
power (int 9, int 3);
```

In the above example, the default value for the first argument is nine and three for second.

```
power p1 (3);
```

In this statement, object p1 is created and nine raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated. Consider the example on the above discussion given below.

```
# include <iostream.h>
# include <conio.h>
# include <math.h>
class power
{
    private:
    int num;
    int power;
    int ans;
    public :
power (int n=9,int p=3); //
declaration of constructor with default argu-ments
    void show( )
    {
        cout <<"\n"<<num <<" raise to "<<power <<" is " <<ans;
    }
};
power :: power (int n,int p )
{
    num=n;
    power=p;
    ans=pow(n,p);
}
main( )
{
    clrscr( );
    class power p1,p2(5);
    p1.show( );
    p2.show( );
    return 0;
}
```

Copy Constructor

One Object Member Variable Values assigned to Another Object Member Variables is called copy constructor.

Syntax

```
class class-name
{
    Access Specifier:
        Member-Variables
        Member-Functions
    public:
        class-name(variable)
        {
            // Constructor code
        }

        ... other Variables & Functions
}
```

Example

```
#include<iostream>
#include<conio.h>
using namespace std;
class Example    {
    // Variable Declaration
    int a,b;
    public:
    //Constructor with Argument
    Example(int x,int y)    {
    // Assign Values In Constructor
    a=x;
    b=y;
    cout<<"\nIm Constructor";
    }
    void Display()    {
    cout<<"\nValues :"<<a<<"\t"<<b;
    }
};

int main()
```

```
{
    Example Object(10,20);
    //Copy Constructor
    Example Object2=Object;
    // Constructor invoked.
    Object.Display();
    Object2.Display();
    // Wait For Output Screen
    getch();
    return 0;
}
```

Dynamic Constructor

This constructor is used to allocate the memory to the objects at the run time. The memory allocation to objects is allocated with the help of 'new' operator.

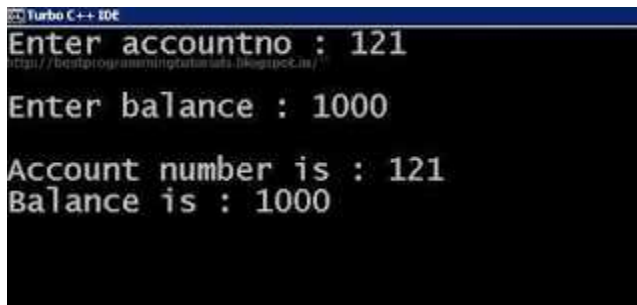
By using this constructor, we can dynamically initialize the objects.

Example :

```
#include <iostream.h>
#include <conio.h>
class Account
{
private:
int account_no;
int balance;
public :
Account(int a,int b)
{
account_no=a;
balance=b;
}
void display()
{
cout<< "\nAccount number is : "<< account_no;
cout<< "\nBalance is : " << balance;
}
};
void main()
{
clrscr();
```

```
int an,bal;
cout<< "Enter account no : ";
cin >> an;
cout<< "\nEnter balance : ";
cin >> bal;
Account *acc=new Account(an,bal); //dynamic constructor
acc->display(); //'-'>' operator is used to access the method
getch();
}
```

Output :

A screenshot of the Turbo C++ IDE. The window title is "Turbo C++ IDE". The output window shows the following text: "Enter accountno : 121", "Enter balance : 1000", "Account number is : 121", and "Balance is : 1000". The background of the output window is black with white text.

Destructor

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>
#include <iostream>
using namespace std;
class myclass {
int a,b;
public:
myclass();    // constructor
~myclass();   // destructor - no need to call the destructor
void show();
};
myclass::myclass()
{
```

Prog.in C++ Laboratory Manual

```
cout << "In constructor\n";
a = 30;
b = 20;
}
myclass::~myclass()
{
//invoked before removing objects from memory
cout << "Destructing...\n";
}
void myclass::show()
{
cout << "A =" << a << endl << "B =" << b << endl;
cout << "Sum is " << a+b << endl;
}
int main()
{
myclass ob;
ob.show();
return 0;
}
```


Exercises

1) Write a C++ program to demonstrate the use of constructor and destructor.

2) Create a class called distance that has a separate integer member data for feet and inches. One constructor should initialize this data to zero and another should initialize it to fixed values. A member function should display it in feet inches format.

3) Write a C++ program to copy the value of one object to another object using copy constructor.

PRACTICAL-IV **INHERITANCE**

Concepts of Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public**, **protected**, or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
    public:
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
}
```

```
};

// Derived class
class Rectangle: public Shape
{
    public:
        int getArea()
        {
            return (width * height);
        }
};

int main(void)
{
    Rectangle Rect;

    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

| Access | public | protected | private |
|-----------------|--------|-----------|---------|
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

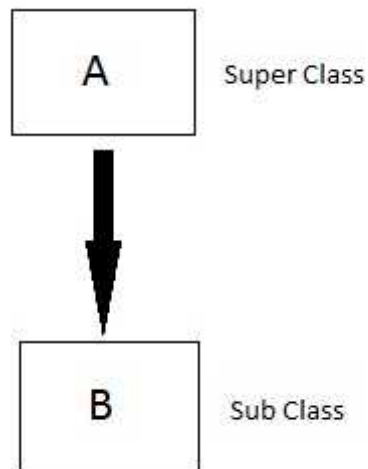
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



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

class student
{
    public:
    int rno;
    //float per;
    char name[20];
    void getdata()
    {
        cout<<"Enter RollNo :- \t";
        cin>>rno;
        cout<<"Enter Name :- \t";
        cin>>name;
    }
};

class marks : public student
```

```
{
public:
    int m1,m2,m3,tot;
    float per;
    void getmarks()
    {
        getdata();
        cout<<"Enter Marks 1 :- \t";
        cin>>m1;
        cout<<"Enter Marks 2 :- \t";
        cin>>m2;
        cout<<"Enter Marks 2 :- \t";
        cin>>m3;
    }
    void display()
    {
        getmarks();
        cout<<"Roll Not \t Name \t Marks1 \t marks2 \t Marks3 \t Total \t
Percentage";

        cout<<rno<<"\t"<<name<<"\t"<<m1<<"\t"<<m2<<"\t"<<m3<<"\t"<<tot<<"\t"
<<per;
    }
};
void main()
{
    student std;
    clrscr();
    std.getmarks();
    std.display();
    getch();
}
```

Making a Private Member Inherited

you've seen the private and public access specifiers, which determine who can access the members of a class. As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class. Note that this means derived classes can not access private members!

```
class Base
{
private:
    int m_nPrivate; // can only be accessed by Base member functions (not derived
classes)
public:
    int m_nPublic; // can be accessed by anybody
};
```

When dealing with inherited classes, things get a bit more complex.

First, there is a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The **protected** access specifier restricts access to member functions of the same class, or those of derived classes.

```
class Base
{
public:
    int m_nPublic; // can be accessed by anybody
private:
    int m_nPrivate; // can only be accessed by Base member functions (but not derived
classes)
protected:
    int m_nProtected; // can be accessed by Base member functions, or derived classes.
};
```

```
class Derived: public Base
{
public:
    Derived()
    {
        // Derived's access to Base members is not influenced by the type of inheritance
used,
        // so the following is always true:

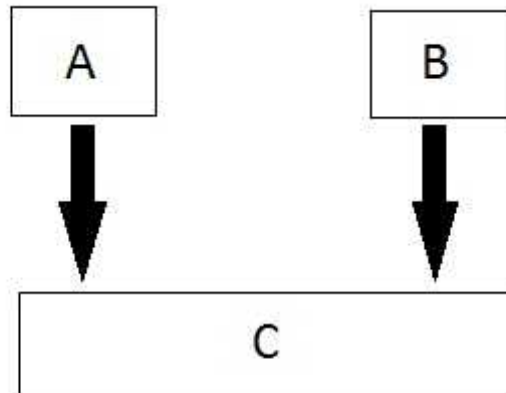
        m_nPublic = 1; // allowed: can access public base members from derived class
        m_nPrivate = 2; // not allowed: can not access private base members from derived
class
        m_nProtected = 3; // allowed: can access protected base members from derived class
    }
};
```



```
};  
  
int main()  
{  
    Base cBase;  
    cBase.m_nPublic = 1; // allowed: can access public members from outside class  
    cBase.m_nPrivate = 2; // not allowed: can not access private members from outside  
class  
    cBase.m_nProtected = 3; // not allowed: can not access protected members from  
outside class  
}
```

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



```
#include<iostream.h>  
#include<conio.h>  
class student  
{  
    protected:  
        int rno,m1,m2;  
    public:  
        void get()  
        {  
            cout<<"Enter the Roll no :";  
            cin>>rno;  
            cout<<"Enter the two marks  :";  
            cin>>m1>>m2;  
        }  
}
```

Prog.in C++ Laboratory Manual

```
};
class sports
{
    protected:
        int sm;          // sm = Sports mark
    public:
        void getsm()
        {
            cout<<"\nEnter the sports mark :";
            cin>>sm;
        }
};
class statement:public student,public sports
{
    int tot,avg;
    public:
    void display()
    {
        tot=(m1+m2+sm);
        avg=tot/3;
        cout<<"\n\n\tRoll No   : "<<rno<<"\n\tTotal   : "<<tot;
        cout<<"\n\tAverage   : "<<avg;
    }
};
void main()
{
    clrscr();
    statement obj;
    obj.get();
    obj.getsm();
    obj.display();
    getch();
}
```

Output:

Enter the Roll no: 100

Enter two marks

90

80

Enter the Sports Mark: 90

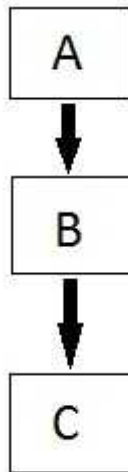
Roll No: 100

Total : 260

Average: 86.66

Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one is sub class for the other.



```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        cout<<"Base class content.";
    }
};
class B : public A
{
};
class C : public B
{
};
```

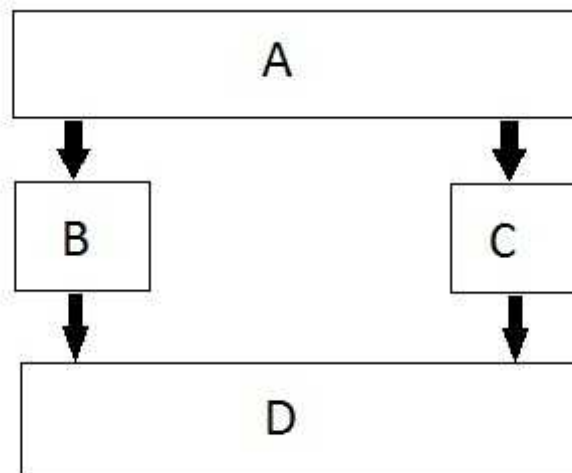
```
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output

Base class content.

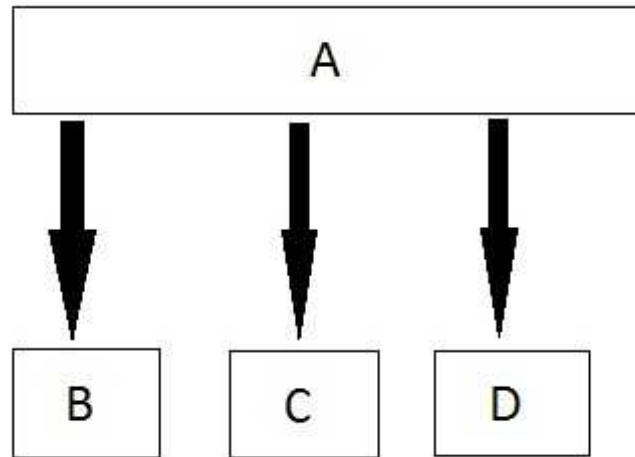
Hybrid Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



Virtual Base Class

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

```
#include<iostream.h>
#include<conio.h>
class base
{
    public:
    virtual void show()
    {
        cout<<"\n Base class show:";
    }
    void display()
    {
        cout<<"\n Base class display:" ;
    }
};
```

Prog.in C++ Laboratory Manual

```
class drive:public base
{
    public:
        void display()
        {
            cout<<"\n Drive class display:";
        }
        void show()
        {
            cout<<"\n Drive class show:";
        }
};
```

```
void main()
{
    clrscr();
    base obj1;
    base *p;
    cout<<"\n\t P points to base:\n" ;
    p=&obj1;
    p->display();
    p->show();
    cout<<"\n\n\t P points to drive:\n";
    drive obj2;
    p=&obj2;
    p->display();
    p->show();
    getch();
}
```

Output:

P points to Base

Base class display

Base class show

P points to Drive

Base class Display

Drive class Show

Abstract Classes

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example of Abstract Class

```
class Base      //Abstract base class
{
public:
virtual void show() = 0;      //Pure Virtual Function
};

class Derived:public Base
{
public:
void show()
{ cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
Base obj;      //Compile Time Error
Base *b;
Derived d;
b = &d;
b->show();
}
```

Output : Implementation of Virtual Function in Derived class

Constructor in Derived Classes

A derived-class constructor must call a base-class constructor before it does anything else. If the base class constructor is not called explicitly in a derived-class constructor, the system will try to invoke the base-class's no-arg constructor. But, remember, a base class will have a no-arg constructor only if you provide one, or, by default, if you have defined no constructors at all for the base class. In what follows, we will first use the User class example to illustrate the more common case, which is that of a derived-class constructor making an explicit call to a base-class constructor.

// program to show how constructors are invoked in derived class

```
#include <iostream.h>
class alpha
(
  private:
    int x;
  public:
    alpha(int i)
    {
      x = i;
      cout << "n alpha initialized n";
    }
    void show_x()
    {
      cout << "n x = "<<x;
    }
);
class beta
(
  private:
    float y;
  public:
    beta(float j)
    {
      y = j;
      cout << "n beta initialized n";
    }
    void show_y()
    {
      cout << "n y = "<<y;
    }
}
```


Prog.in C++ Laboratory Manual

```
);  
class gamma : public beta, public alpha  
{  
private:  
    int n,m;  
public:  
    gamma(int a, float b, int c, int d):: alpha(a), beta(b)  
    {  
        m = c;  
        n = d;  
        cout << "n gamma initialized n";  
    }  
    void show_mn()  
    {  
        cout << "n m = "<<m;  
        cout << "n n = "<<n;  
    }  
};
```

```
void main()  
{  
    gamma g(5, 7.65, 30, 100);  
    cout << "n";  
    g.show_x();  
    g.show_y();  
    g.show_mn();  
}
```

Output:

beta initialized

alpha initialized

gamma initialized

x = 5

y = 7.65

m = 30

n = 100

Exercises

1) Define a class called student that has the following data members:

- int student number
- string student name
- double student average

The following member functions:

- Constructor that initialize the data members with default values.
- set and get functions for each data member
- Print function to print the values of data members.

Define a class called graduatestudent that inherits data members and functions from the class student, and then declare the following data members :

- int level
- int year

Member functions:

- constructor -set and get functions for each data member
- Print function.

Define a class called master that inherits data members and functions from graduatestudent class, and then declare the following data member:

- int newid.

Member function:

- constructor
- set and get function for the data member
- Print function.

Write a driver program that:

- Declare object of type student with suitable values then print it
- Declare object of type master with your information then print it.

2) Write a C++ program to demonstrate multiple inheritances

3) Write a C++ program to demonstrate constructor call in the derived class

PRACTICAL-V

POINTERS,VIRTUAL FUNCTIONS AND POLYMORPHISM

Pointers to objects,

A variable that holds an address value is called a pointer variable or simply pointer.

Pointer can point to objects as well as to simple data types and arrays.

Sometimes we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. new returns a pointer to an unnamed objects. Let's see the example

```
using namespace std;

class Box
{
public:
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
    }
    double Volume()
    {
        return length * breadth * height;
    }
private:
    double length;    // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};

int main(void)
{
    Box Box1(3.3, 1.2, 1.5); // Declare box1
    Box Box2(8.5, 6.0, 2.0); // Declare box2
    Box *ptrBox;           // Declare pointer to a class.

    // Save the address of first object
    ptrBox = &Box1;
```

```
// Now try to access a member using member access operator
cout << "Volume of Box1: " << ptrBox->Volume() << endl;

// Save the address of first object
ptrBox = &Box2;

// Now try to access a member using member access operator
cout << "Volume of Box2: " << ptrBox->Volume() << endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

'this' Pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, **this** may be used to refer to the invoking object. Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Let us try the following example to understand the concept of this pointer:

```
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};
```



```
int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output:

```
x = 20
```

Pointer to Derived Classes

Inheritance provides hierarchical organization of classes. It also provides hierarchical relationship between two objects and indicates the shared properties between them. All derived classes inherit properties from the common base class. Pointers can be declared to point base or derived classes. Pointers to object of base class are type compatible with pointers to object of derived class. A base class pointer can also point to objects of base and derived classes. In other words, a pointer to base class object can point to objects of derived classes whereas a pointer to derived class object cannot point to objects of base class object.

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

```
class BaseClass {
    int x;
public:
    void setX(int i) {
        x = i;
    }
    int getX() {
        return x;
    }
};
```

```
class DerivedClass : public BaseClass {
    int y;
public:
    void setY(int i) {
        y = i;
    }
    int getY() {
        return y;
    }
};
```

```
};

int main()
{
    BaseClass *p;           // pointer to BaseClass type
    BaseClass baseObject;  // object of BaseClass
    DerivedClass derivedObject; // object of DerivedClass

    p = &baseObject;       // use p to access BaseClass object
    p->setx(10);            // access BaseClass object
    cout << "Base object x: " << p->getx() << "\n";

    p = &derivedObject;    // point to DerivedClass object
    p->setx(99);            // access DerivedClass object

    derivedObject.sety(88); // can't use p to set y, so do it directly
    cout << "Derived object x: " << p->getx() << "\n";
    cout << "Derived object y: " << derivedObject.gety() << "\n";

    return 0;
}
```

Virtual Functions

If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as polymorphism which is one of the important feature of OOP.

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable. This can be demonstrated by an example.

In order to make a function virtual, you have to add keyword **virtual** in front of a function.

```
/* Example to demonstrate the working of virtual function in C++
programming. */

#include <iostream>
```

```
using namespace std;
class B
{
    public:
        virtual void display()    /* Virtual function */
        { cout<<"Content of base class.\n"; }
};

class D1 : public B
{
    public:
        void display()
        { cout<<"Content of first derived class.\n"; }
};

class D2 : public B
{
    public:
        void display()
        { cout<<"Content of second derived class.\n"; }
};

int main()
{
    B *b;
    D1 d1;
    D2 d2;

    /* b->display(); // You cannot use this code here because the function of base
    class is virtual. */

    b = &d1;
    b->display(); /* calls display() of class derived D1 */
    b = &d2;
    b->display(); /* calls display() of class derived D2 */
    return 0;
}
```

Output

Content of first derived class.

Content of second derived class.

Exercises

1) Write a program in C++ to demonstrate virtual function

2) Write a C++ Program that shows use of this pointer

PRACTICAL-VI

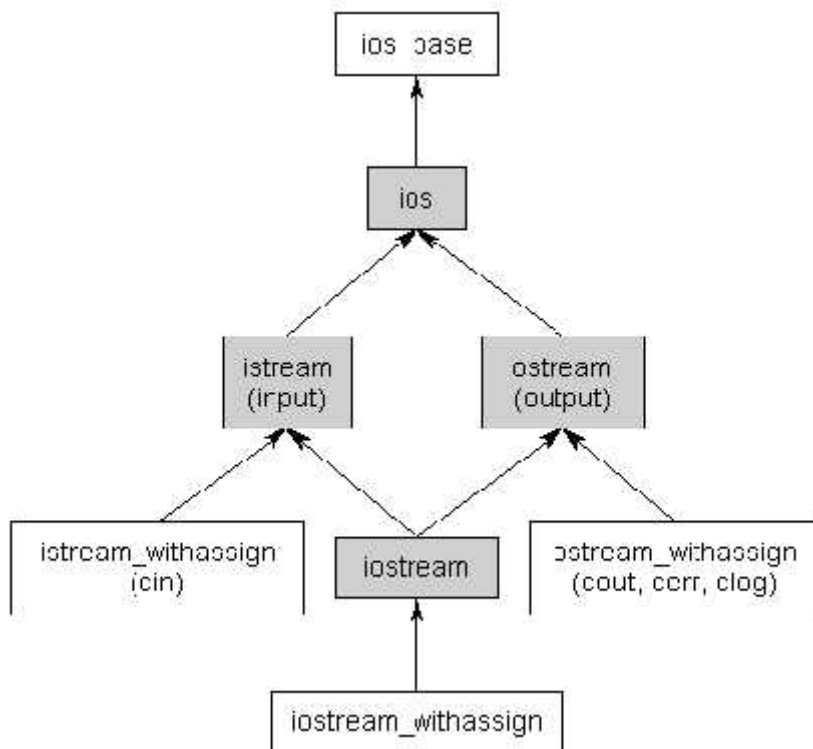
MANAGING CONSOLE I/O OPERATIONS

Input and Output Streams

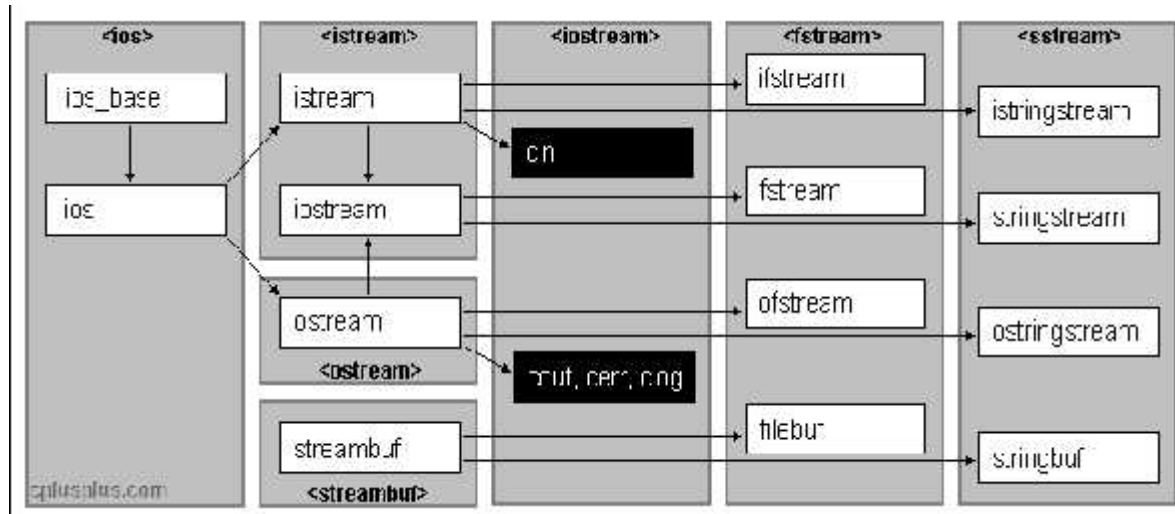
Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the std namespace). In previous lessons, you included the iostream library header and made use of the cin and cout objects to do simple I/O. In this lesson, we'll take a look at the iostream library in more detail.

The iostream library

When you include the iostream header, you gain access to a whole heirarchy of classes responsible for providing I/O functionality (including one class that is actually named iostream). The class heirarchy for the non-file-I/O classes looks like this:



Input/Output library



Unformatted and formatted I/O Operations

The input stream uses cin object to read data and the output stream uses cout object to display data on the screen. The cin and cout are predefined streams for input and output of data. The data type is identified by these functions using operator overloading of the operators <<(insertion operator) and >> (extraction operator). The operator << is overloaded in the ostream class and the operator >> is overloaded in istream class. Figure 2.7 shows the flow of input and output stream.

INPUT STREAM The input stream does read operation through keyboard. It uses cin as object. The cin statement uses >> (extraction operator) before variable name. The cin statement is used to read data through the input device.

Get() and Put() function

Single Character input and output operations in C++ can be done using get() and put() functions.

The get() function is used to read a character and put() function is used to display a character on a screen. The get() function has two syntaxes

1. `get(void);`
`get(char *);`

The put() function is used to display a character on screen. The syntax of put() is

```
Cout.put('A');
```

Example

```
#include <iostream.h>

using namespace std;

int main()

{

int count = 0;

char c;

cout << "INPUT TEXT \n ";

cin.get (c);

while (c !='\n')

{

cout.put (c);

count++;

cin.get ( c);

}

cout << "\n Number of characters = " << count << "\n";

return 0;

}
```

Getline() function

The getline() function reads the string including spaces. The cin function does not allow the string to enter blank spaces. The input reading is terminated when user presses the enter key. The new line character is accepted but not saved and is replaced with null character.

```
Cin.getline(variable,size);
```


Read() and write() function

The read() function is used to read text through the keyboard .When we se read statement it is necessary to enter character equal to the number of size specified .The syntax is

```
Cin.read(variable,size);
```

The write() function is used to display the string on the screen.Its format is same as getline() but the function is exactly opposite .Cout.write()displays only specified number of characters given in the second argument,though actual string may be more in length.The syntax is

```
Cout.write(variable,size);
```

Example of getline()

```
// istream::getline example
#include <iostream> // std::cin, std::cout

int main () {
    char name[256], title[256];

    cout << "Please, enter your name: ";
    cin.getline (name,256);

    cout << "Please, enter your favourite movie: ";
    cin.getline (title,256);

    cout << name << "'s favourite movie is " << title;

    return 0;
}
```

Example of write()

```
#include <iostream>

int main ()
{
    char * string1= "c++";
    char * string2= "Programming";
    int m =strlen (string1);
    int n =strlen (string2);
    for (int i=1 ;i <n ; i++)
    {
        cout.write(string2,i);
    }
}
```

```
        cout<<"\n";
    }
    for (i=n ;i>0 ; i--)
    {
        cout.write(string2,i);
        cout<<"\n";
    }
cout.write(string1,m).write(string2,n);
cout << "\n";
cout.write(string1,10);
return 0;
}
```

Formatting with Manipulators

C++ provides tools to format output. These format manipulators are available by including the file <iomanip.h> as a compiler directive at the top of the source code.

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

setw (width) manipulator

Causes the next value to be displayed in a field specified by width. This setw (width) command is placed in the output expression prior to the output of variables or text constants.

Code:

```
cout << setiosflags(ios::right);
cout << "1234567890" << endl;
cout << setw(10) << "abcde" << endl;    //setting 10 columns, items are printed right
justified - only valid for this line
cout << 45 << endl;    //the default is left justified
```

Run output:

```
1234567890
   abcde
45
```

The text constant "abcde" will be right justified in a field-width of 10 columns. However, the setw (10) does not affect the next output.

The setw (width) manipulator only applies to the next value to be printed. However, the other manipulators will affect the appearance of all values which follow.

setprecision

controls the number of decimal points displayed by a floating point value. The default precision display setting is 6 decimal places to the right of the decimal point.

Code:

```
cout << 321.23456789 << endl;
    // default, prints 6 places to right of decimal
cout << setprecision (4);
cout << 321.2345678 << endl;    // displays 4 places to right of decimal
cout << 7.28 << endl;        // only 7.28 gets printed, not enough places
cout << 25 << endl;          // setprecision () has no effect on integers
```

Run output:

```
321.234568
321.2346
7.28
25
```

setiosflags manipulator

Provides for many options of which one or more can be used at a time. Multiple options are separated by the (|) symbol. Some of the options available are:

| Option | Description |
|-----------------|-------------------------------------------------------------------|
| ios:showpoint | Displays decimal point and trailing zeros as necessary |
| ios::fixed | Displays real values in fixed-point form. Values are rounded off. |
| ios: scientific | Displays real values in floating-point form |
| ios::left | Displays values left-justified |
| ios::right | Displays values right-justified within a field |

Example 3: ios::showpoint

Code:

```
cout << setiosflags (ios::showpoint);    // will cause trailing zeros to print
cout << setprecision (4);
cout << 1.25 << endl;
```

Run output:

1.2500

Example 4: ios::scientific with setprecision ()

Code

```
cout << setprecision (2) << setiosflags (ios::scientific);
cout << 1.25 << endl;
cout << 12365.6219 << endl;    // answer is truncated to 2 decimal places
cout << 6.023e23 << endl;
```

Run output:

1.25e+00

1.23e+04

6.02e+23

Example 5: ios::left

Code:

```
cout << "12345678901234567890" << endl;
cout << setiosflags (ios::left) << setw (10) << "abcde";
// left-justify state is still turned on
cout << 45 << endl;
```

Run output:

12345678901234567890

abcde 45

Example 6: ios::right, ios::showpoint, ios::fixed, and setprecision()

Code:

```
cout << setiosflags (ios::right | ios::fixed | ios::showpoint);
cout << setprecision (2);
cout << setw(10) << 20 << endl;
cout << setw(10) << 25.95 << endl;
cout << setw(10) << 123.456 << endl;
```

Run output:

20.00

25.95

123.46

Exercise

1) Write a C++ Program to format output with manipulators.

2) Write a C++ program to read a list containing item name, item code and cost interactively and display the data in a tabular format as shown below:

| NAME | CODE | COST |
|------|------|------|
|------|------|------|

3) Design your own manipulator to provide the following output specification for printing money value:

- 1) 10 columns width**
- 2) The character '\$' at the beginning**
- 3) Showing '+' sign.**
- 4) Two digits precision**
- 5) Filling of unused spaces with '*'**
- 6) Trailing zeros shown**