FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -1 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

https://hemanthrajhemu.github.io

MODULE 1 INTRODUCTION TO OBJECT ORIENTED CONCEPTS

The following concepts should be learn in this Module

A Review of structures, Procedure–Oriented Programming system, Object Oriented, Programming System, Comparison of Object Oriented Language with C, Console I/O, variables and reference variables, Function Prototyping, Function Overloading. Class and Objects: Introduction, member functions and data, objects and functions.

Text book 1: Ch 1: 1.1 to 1.9 Ch 2: 2.1 to 2.3 , RBT: L1, L2

A REVIEW OF STRUCTURES

Consider a function nextday( ) that accepts the addresses of 3 integers that represent a date and changes these values to represent nextday.

Prototype of this function //for calculating the next day
Suppose

If we call
d=1; m=1;
y=2002; //1st january 2002
.

Members of wrong group may be accidentally sent to the function

```
d1=28; m1=2; y1=1999;         //28thfeb99
d2=19; m2=3; y2=1999;         //19thmrch99
nextday(&d1,&m1,&y1);         //ok
nextday(&d1,&m2,&y2); //incorrect set passed
```

There is nothing in language itself that prevents the wrong set of variables from being sent to the function.
So we need structures

1) Putting structure definition and prototypes of associated functions in a header file(date.h)

```
Struct date
{
int d, m, y;
};
Void nextday(struct date *);
```

2) Put definition and other prototypes in a source code and create a library

https://hemanthrajhemu.github.io

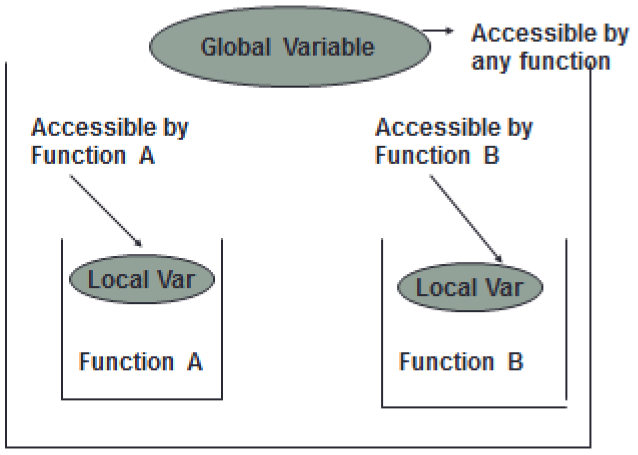
```
void main()
{
Struct date d1;
d1.d=28;
d1.m=2;        //Need for structures
d1.y=1999;
nextday(&d1);
}
```

3) Using Structures in Application Programs

- Include header file provided by programmer in the source code.
- Declare variables of new data type in the source code
- Embed calls to the associated functions by passing these variables in the source code

PROCEDURE/ STRUCTURE ORIENTED PROGRAMMING

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP).
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.
- The primary focus is on functions.
- Dividing a program into functions and modules is one of the cornerstones of Structured Programming.
- Since many functions in a program can access global data / global variables, global data can be corrupted by that have no business to change it.
- Hence we need a way to restrict the access to the data, to hide it from all but a few critical functions. This protects the data, simplifies the maintenance and other benefits.



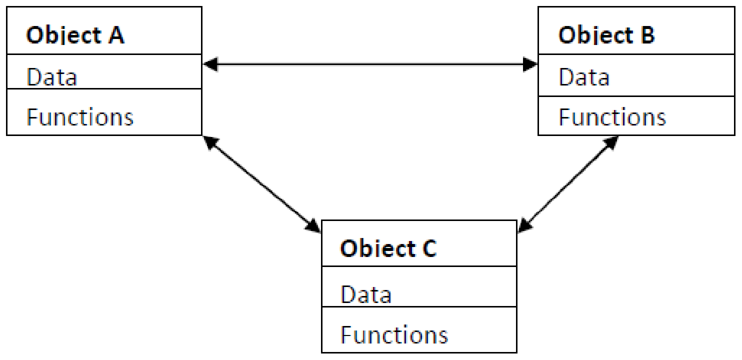
https://hemanthrajhemu.github.io

Drawback/Disadvantage

1. Data is not secure and can be manipulated by any function/procedure.
2. Associated functions that were designed by library programmer don‟t have rights to work upon the data.
3. They are not a part of structure definition itself because application program might modify the structure variables by some code inadvertently written in application program itself

OBJECT ORIENTED PROGRAMMING

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions(methods).
- An object functions are called Member functions in C++.
- Data and functions are said to be encapsulated in an object.
- Data Encapsulation & Data Hiding are the key terms in OOPs



OOPs simplify writing, debugging and maintaining the program. C++ program typically contain a number of objects interacting with each other by calling one another‟s member functions.

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.

https://hemanthrajhemu.github.io

Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Objects

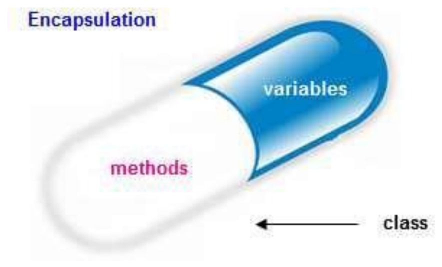
Objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

Class

Object contains data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

Data Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation.
- The data is not accessible to the outside world, only those function which are wrapped in the class can access it.
- These functions provide the interface between the object"s data and the program.
- It hides the implementation details of an object from its users.
- Encapsulation prevents unauthorized access of data or functionality.
- This insulation of the data from direct access by the program is called **data hiding** or **information hiding**.



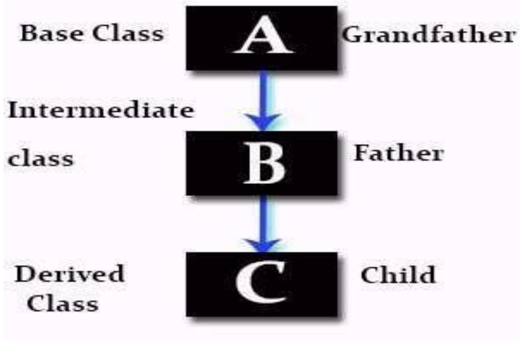
https://hemanthrajhemu.github.io

Data Abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Since classes use the concept of data abstraction, they are known as **Abstract Data Types(ADT)**.
- It separates unnecessary details or explanations so as to reduce complexities of understanding requirements.

Inheritance

- **Inheritance** is the process by which objects of one class acquire the properties of objects of another class.
- Parent class can be given the general characteristics, while its child may be given more specific characteristics.
- Inheritance mechanism makes it possible for one object to be a specific instance of a more general class.
- Reusability – adding additional features to an existing class without modifying it. That is, deriving a new class from the existing one. The new class will have the combined features of both the classes. New class is also called as **Derived** class and existing class is called as **Base** Class.



Polymorphism

- **Polymorphism,** a Greek term means to ability to take more than one form.
- An operation may exhibits different behaviors in different instances. The behavior depends upon the type of data used in the operation.. **Ex: Function Overloading**
- For example consider the operation of addition for two numbers; the operation will generate a sum. If the operands are string then the operation would produce a third string by concatenation.
- The process of making an operator to exhibit different behavior in different instances is known **operator overloading.**

C	C++
1. C compiler cannot execute C++ programs	**1. C++ compiler can execute C programs**
2. In C, u may / may not include function prototypes	**2. In C++, you must Include function prototypes**
3. C doesn't allow for default arguments	**3. C++ lets you to specify default arguments in function prototype**
4. Declaration of the variables must be at the beginning	**4. Declaration of the variables can be anywhere before using**

https://hemanthrajhemu.github.io

5. If a C program uses a Local variable that has Same name as global variable, then C uses the value of a local variable.	**5. In C++, u can instruct program to use value of global variable with scope resolution Ex- cout << “Iamglobal var :” << ::I;**
6. Fun overloading is not there.	**6. Fun overloading exists**
7. Function inside the structure is not allowed	**7. Function inside the structure is allowed**
8. Object initialization doesn’t exist	**8. Object initialization (constructor) exist**
9. Data hiding, data abstraction and data encapsulation feature doesn’t exist	**9. Data hiding, data abstraction and data encapsulation exists in C++**

CONSOLE INPUT AND OUTPUT

Console output

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.

```
cout << constant/variable
cout<<endl;
cout<<“\n”;\\newline
```

Console input

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.

```
cin>> variable
```

Example : #include <iostream.h>

```
void main ()
{
 int x;
 cout << "Please enter an integer value: ";
 cin >>x;
 cout <<endl<< "Value you entered is " <<x;
 cout << " and its double is " <<x*2 << ".\n";
}
```

Variables in C++

- Can be declared anywhere in the C++ program before using them.

Reference variable

- They are used as aliases for other variables within a function.
- All operations supposedly performed on the alias (i.e., the reference) are actually

https://hemanthrajhemu.github.io

performed on the original variable.

- An alias is simply another name for the original variable.
- Must be initialized at the time of declaration.
- Reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.

Syntax :

datatype &variable= existing variable

Example

Ex : #include <iostream.h>
void main()
{
  int x=3;
  int &y=x;
  cout<<“x=“<<x endl<<“y=“<<y<<endl;
  y=7;
  cout<<“x=“<<x<<endl<<“y=“<<y<<endl;
 }
 	
Output :
x=3
y=3
x=7
Y=7
	Ex: #include <iostream.h>
	void main()
	{
	   int x, y;
	   int x=100;
	   int & iRef=x;
	   y=iRef;
	   cout <<y<<endl;
	   y++; // x and iRef unchanged
	   cout <<x <<endl<<iRef<<endl<<y<<endl;
	 }
	100
	100
	100
	101

#include <iostream>
using namespace std;
int main ()
{
int i; double d;
int& r = i;
double& s = d;
i = 5;
cout << "Value of i : " << i << endl;
cout << "Value of i reference : " << r << endl;
d = 11.7;
cout << "Value of d : " << d << endl;
cout << "Value of d reference : " << s << endl;
return 0;
}
	Output:
	Value of i : 5
	Value of i reference : 5
	Value of d : 11.7
	Value of d reference : 11.7

Swap 2 variables using reference variables

https://hemanthrajhemu.github.io

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);

int main () {
      int a = 100;
      int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
```

```
     swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
void swap(int &x, int &y) {
   int temp;
               temp = x;
               x = y;
               y = temp;
}
```

FUNCTION PROTOTYPING

- C++ strongly supports function prototypes
- Prototype describes the function‟s interface to the compiler
- Tells the compiler the return type of function, number , type and sequence of its formal arguments

Syntax : **return_type function_name(argument_list);**

Eg- int add (int, int);

With prototyping, compiler ensures following .The return value of a function is handled correctly. Correct number and type of arguments are passed to a function. Since C++ compiler requires function prototyping, it will report error against function call because no function prototype is provided to resolve the function call. Prototyping guarantees protection from errors arising out of incorrect function calls. A function heading without body.

https://hemanthrajhemu.github.io

FUNCTION OVERLOADING IN C++

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading. However to achieve this they must have different signatures.
Signature of a function means number, type and sequence of formal arguments of the function.

Ways to overload a function

- By changing number of Arguments.
- By having different types of argument.

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

void display(int);
void display(float);
void display(int, float);
```

```

int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);
   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}

void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

https://hemanthrajhemu.github.io

INLINE FUNCTIONS

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

For an inline function, declaration and definition must be done together. For example,

Syntax:

```
inline return_type function_name(arguments)
{
//function body
}
```

Example:

```
inline double cube(double a)
{
return(a * a * a );
}
```

Why to use –

In many places we create the functions for small work/functionality which contain simple and less number of executable instruction. Imagine their calling overhead each time they are being called by callers. With inline keyword, 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.

CLASS AND OBJECTS

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. A class declaration is similar syntactically to a structure.

General form of a class declaration is:

https://hemanthrajhemu.github.io

class class_name
{
private:
Variable declaration/data members;
Function declaration/ member functions;
protected:
Variable declaration/data members;
Function declaration/ member functions;
public:
Variable declaration/data members;
Function declaration/ member functions
};
	Private members can be accessed only from within the class.
	
	Protected members can be accessed by own class and its derived classes.
	
	Public members can be accessed from outside the class also.

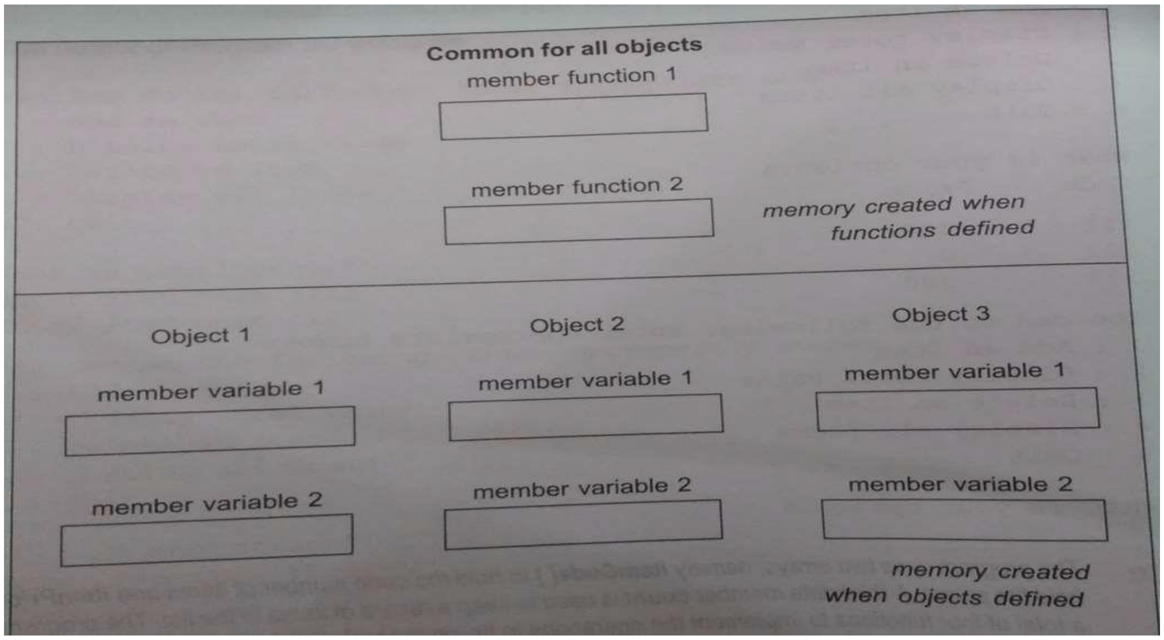
- The variables declared inside the class definition are known as **data members** and the functions declared inside a class are known as **member functions**.
- By default the data members and member function of a class are **private**.
- Private data members can be accessed by the functions that are wrapped inside the class.

How to access member of a class?

To access member of a class dot operator is used. i.e.

object-name . data-member ***and***
object-name . member-function

Memory allocation of objects:



https://hemanthrajhemu.github.io

Memory space for objects is allocated when they are declared and not when the class is specified. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.
Only space for member variables(data members) is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

General steps to write a C++ program using class and object:

- Header files
- Class definition
- Member function definition
- void main function

ex:
```
#include<iostream.h>
        class Add
        {
        int x, y, z;
        public:
         void getdata()
        {
        cout<<"Enter two numbers";
        cin>>x>>y;
        }
        void calculate(void);
        void display(void);
        };
```

```
                void Add :: calculate()
                {
                z=x+y;
                }

                void Add :: display()
                {
                cout<<z;
                }
        void main()
        {
             Add a;
               a.getdata();
               a.calculate();
               a.display();
        }
```

https://hemanthrajhemu.github.io

Output:

```
Enter two numbers 5 6
11
```

Private & Public Members

- The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.

```
class Student
{
 private:    // private data member
 int rollno;

 public:     // public accessor functions
 int getRollno()
 {
  return rollno;
 }

 void setRollno(int i)
 {
  rollno=i;
 }

};

int main()
{
 Student A;
 A.rollono=1;                    //Compile time error
 cout<< A.rollno;                //Compile time error

 A.setRollno(1);                    //Rollno initialized to 1
 cout<< A.getRollno();              //Output will be 1
}
```

- The public keyword makes data and functions public. Public data and functions can be accessed out of the class.

https://hemanthrajhemu.github.io

```
class Student
{
 public:
         int rollno;
 string name;
};

int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";

 B.rollno=2;
 B.name="Bella";

 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

Scope Resolution Operator (::)

It is possible and necessary for Library programmer to define member functions outside their respective classes.

The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is

```
Return_type class_name :: function_name (parameter_list)
{
// body of the member function
}
```

In C++, scope resolution operator is **::**. It is used for following purposes.

1) **To access a global variable when there is a local variable with same name**
2) **To define a function outside a class.**

https://hemanthrajhemu.github.io

3) **To access a class's static variables. [refer static data members]**
4) **In case of multiple Inheritance**

Global variable access:

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

int x; // Global x

int main()
{
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Define function outside the class

```
class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box
      double getVolume(void);
       void setdata( double l, double b, double h );
};
            double Box:: getVolume(void)
             {
                    return length * breadth * height;
                          }

void Box::setLength(double l, double b, double h)
 {
   length = l;
breadth = b;
height = h;
       }

int main( )
 {
   Box Box1;
double volume = 0.0;
Box1.setdata(6.0,5.0,8.3);

volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
  }
```

https://hemanthrajhemu.github.io

This pointer

The „this‟ pointer-The facility to create and call member functions of class objects is provided by the compiler. Compiler does this by using a unique pointer -> this.
this pointer - always a constant pointer. It points at the object with respect to which the function was called.
The „this‟ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. „this‟ pointer is a constant pointer that holds the memory address of the current object. „this‟ pointer is not available in static member functions as static member functions can be called without any object (with class name).

- It puts a declaration of the this pointer as a leading formal argument in the prototypes of all member functions as follows

```
void setFeet( Distance * const this, int x)
{
        this -> iFeet = x;
  }
```

It passes the address of invoking object as a leading parameter to each call to the member functions.
Ex : setFeet(**&d1,** 1);

```
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

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

https://hemanthrajhemu.github.io

MEMBER FUNCTIONS AND MEMBER DATA

1) Default values for formal arguments of Member functions

Default values can be assigned to arguments of non-member functions and member functions. Member functions should be overloaded with care, if default values are specified for some or all of its arguments otherwise the compiler will report ambiguity error.

Ex

```
Class A
  {
    int sum(int x, int y, int z=0)
    {
        return (x + y + z );
    }
 };
    int main()
    {
       A A1;
      A1. sum(10, 15) << endl;
      A1. sum(10, 15, 25) << endl;
      return 0;
     }
```

```
Output: 25
        50
```

Points to remember

- Default values are specified from right to left.
- Default values must be specified in the function prototypes and not in function definitions.
    - **int mul(int i, int j=5, int k=10);   // Valid**
    - **int mul(int i=5, int j);   //InValid**
    - **int mul(int i=0, int j, int k=10);   // InValid**
    - **int mul(int i=2, int j=5, int k=10);   // Valid**

https://hemanthrajhemu.github.io

2) Constant Member functions:

A function becomes const when const keyword is used in function"s declaration. The idea of const functions is not allow them to modify the object on which they are called.

The programmer desires that one of the member functions of a class should not be able to change the value of data members. This function should be able to merely read the values contained in the data members, but not change them.

```
Class Distance
{
   int iFeet;
   float fInches;
         public:
                 void setdata(int,float);
                 void getdata() const;            //constant function
         };

          void Distance::setdata(int x, float y)
          {
              iFeet=x;
             fInches=y;
           }
          void Distance::getdata()      const            //const function
           {
               iFeet++;                  //ERROR!!
              fInches=fInches+ 1;       //ERROR!!
              Cout<<iFeet;
            }
```

3) Mutable Data Members

Mutable data member is never constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the key word mutable makes it mutable.

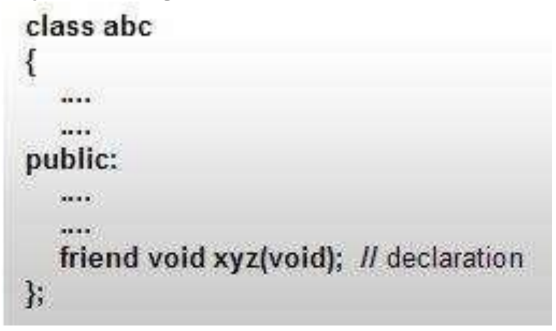
```
Class A
{
                int  x;
                mutable int y;
        public:
                void abc() const              //a constant member function
                {
                        x++;          // error: cant modify a non-constant
```

https://hemanthrajhemu.github.io

```
                y++;        //ok can modify a mutable data member in
            }
            void def()     //can modify
            {
                x++;
                y++;
            }
};
```

4) Friend Class & Friend Functions:

Friend Function

- Scope of a friend function is not inside the class in which it is declared. It is prefixed with the keyword **friend.**
- Since its scope is not inside the class, it cannot be called using the object of that class. It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator. A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend
- It can be declared either in private or public part of the class definition. Usually it has the objects as arguments.
- 

```
class abc
{
    ....
    ....
public:
    ....
    ....
    friend void xyz(void);  // declaration
};
```

```
class Add
{
int x, y, z;
public:
Add(int, int);
friend int calculate(Add p);
};

Add :: Add(int a, int b)
{
x=a;
y=b;
}
```

https://hemanthrajhemu.github.io

```
int calculate(Add p)
{
return(p.x+p.y);
}

void main()
{
Add a(5, 6);
cout<<calculate(a);
}
```

Output:

11

<u>Friend Class:</u> A class can be friend of another class. Member Functions of a friend class can access private data members of objects of class of which it is a friend**.** When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
class B;                  //Forward declaration.
 class A
    {
       friend class B;
    //rest of the class A
     };
```

Friendship is not transitive. That is, If class A is friend with class B, and class B is friend with class C. This doesn't mean that class A is friend with class C.

```
#include <iostream>
using namespace std;
class Rectangle;
class Square
{
        friend class Rectangle;      // declaring Rectangle as friend class
        int side;
        public:
                Square ( int s )
                {
                        side = s;
                }
};
```

https://hemanthrajhemu.github.io

```
class Rectangle
{
        int length;
        int breadth;
        public:
        int getArea()
        {
                return length * breadth;
        }
        void shape( Square a )
        {
                length = a.side;
                breadth = a.side;
        }
};

int main()
{
        Square square(5);
        Rectangle rectangle;
        rectangle.shape(square);
        cout << rectangle.getArea() << endl;
        return 0;
}
```

Output:

```
25
```

5) Static Data Members

- The data member of a class preceded by the keyword **static** is known as static member.
- When we precede a member variable's declaration with static, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable.
- All static variables are initialized to zero before the first object is created.

```
class A
{
    int  p;
    static int q;
public:
```

https://hemanthrajhemu.github.io

```
            A();
            void incr(void);
            void display(void);
};

    A :: A()
        {
          p=5;
        }
int A:: q=10;           // initialize static data member

void A:: incr()
      {
             p++;
              q++;
        }

void A:: display()
        {
          cout<<p<<"\t"<<q<<endl;
         }
void main()
{
A a1, a2, a3;
a1.incr();
a1.display();
a2.incr();
a2.display();
a3.incr();
a3.display();
}
```

Output:

```
6 11
6 12
6 13
```

Static Member function/method

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member Function.). It is accessed by class name and not by object‟s name .

  i.e. **class-name::function-name**

- The function name is preceded by the keyword **static**. A static member function does not have this pointer.

https://hemanthrajhemu.github.io

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

class Counter
{
        private:
            static int count;               //static data member as count

        public:
                Counter()                   //default constructor
                {
                    count++;
                  }
        static void Print()                      //static member function
                {
                        cout<<"\nTotal objects are: "<<count;
                }
};

int Counter :: count = 0;              //count initialization with 0

int main()
{
        Counter OB1;
        OB1.Print();

        Counter OB2;
        OB2.Print();

        Counter OB3;
        OB3.Print();

        return 0;
}
```

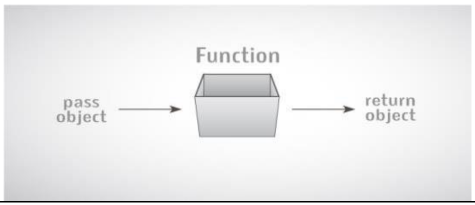
Output:

```
Total objects are: 1
Total objects are: 2
Total objects are: 3
```

https://hemanthrajhemu.github.io

OBJECTS & FUNCTIONS:

Objects appear as local variables. They can also be passed by reference to Functions. Finally they can be returned by value or by reference from the functions.



```
#include  <iostream>
using namespace std;
class Complex
{
   private:
                int real;
                int imag;
   public:
                void readData()
                {
                        cout << "Enter real and imaginary number
respectively:"<<endl;
                         cin >> real >> imag;
                 }

                 void addComplexNumbers(Complex comp1, Complex comp2)
                {
                          real=comp1.real+comp2.real;
                          imag=comp1.imag+comp2.imag;
                 }
      void displaySum()
      {
         cout << "Sum = " << real<< "+" << imag << "i";
      }
  };

int main()
{
   Complex c1,c2,c3;
   c1.readData();
   c2.readData();
   c3.addComplexNumbers(c1, c2);
   c3.displaySum();
   return 0;
}
```

Output:

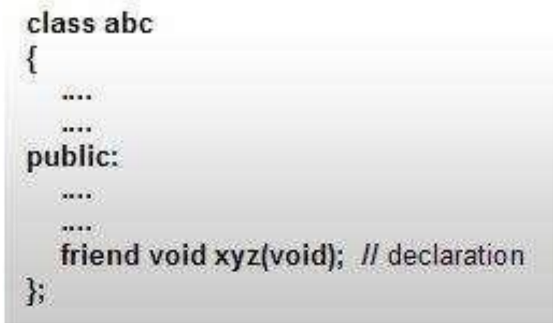
```
1
2
1
2
Sum= 2+4i
```

https://hemanthrajhemu.github.io

```
                        y++;          //ok can modify a mutable data member in
                }
                void def()      //can modify
                {
                        x++;
                        y++;
                }
};
```

6) Friend Class & Friend Functions:

Friend Function

- Scope of a friend function is not inside the class in which it is declared. It is prefixed with the keyword **friend.**
- Since its scope is not inside the class, it cannot be called using the object of that class. It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator. A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend
- It can be declared either in private or public part of the class definition. Usually it has the objects as arguments.
- 

```
class abc
{
   ....
   ....
public:
   ....
   ....
   friend void xyz(void);  // declaration
};
```

```
class Add
{
int x, y, z;
public:
Add(int, int);
friend int calculate(Add p);
};

Add :: Add(int a, int b)
{
x=a;
y=b;
}
```

https://hemanthrajhemu.github.io

```
int calculate(Add p)
{
return(p.x+p.y);
}

void main()
{
Add a(5, 6);
cout<<calculate(a);
}
```

Output:
11

Friend Class: A class can be friend of another class. Member Functions of a friend class can access private data members of objects of class of which it is a friend**.** When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
class B;                //Forward declaration.
 class A
    {
       friend class B;
    //rest of the class A
     };
```

Friendship is not transitive. That is, If class A is friend with class B, and class B is friend with class C. This doesn't mean that class A is friend with class C.

```
#include <iostream>
using namespace std;
class Rectangle;
class Square
{
        friend class Rectangle;     // declaring Rectangle as friend class
        int side;
        public:
                Square ( int s )
                {
                        side = s;
                }
};
```

https://hemanthrajhemu.github.io

```
class Rectangle
{
        int length;
        int breadth;
        public:
        int getArea()
        {
                return length * breadth;
        }
        void shape( Square a )
        {
                length = a.side;
                breadth = a.side;
        }
};

int main()
{
        Square square(5);
        Rectangle rectangle;
        rectangle.shape(square);
        cout << rectangle.getArea() << endl;
        return 0;
}
```

Output:

```
25
```

7) Static Data Members

- The data member of a class preceded by the keyword **static** is known as static member.
- When we precede a member variable's declaration with static, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable.
- All static variables are initialized to zero before the first object is created.

```
class A
{
     int  p;
     static int q;
public:
```

https://hemanthrajhemu.github.io

```
          A();
          void incr(void);
          void display(void);
};

    A :: A()
        {
          p=5;
        }
int A:: q=10;          // initialize static data member

void A:: incr()
      {
           p++;
            q++;
       }

void A:: display()
       {
         cout<<p<<"\t"<<q<<endl;
        }
void main()
{
A a1, a2, a3;
a1.incr();
a1.display();
a2.incr();
a2.display();
a3.incr();
a3.display();
}
```

Output:

```
6 11
6 12
6 13
```

Static Member function/method

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member Function.). It is accessed by class name and not by object‟s name .

  i.e. **class-name::function-name**

- The function name is preceded by the keyword **static**. A static member function does not have this pointer.

https://hemanthrajhemu.github.io

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

class Counter
{
        private:
            static int count;              //static data member as count

        public:
                Counter()                  //default constructor
                {
                    count++;
                  }
        static void Print()                    //static member function
                {
                        cout<<"\nTotal objects are: "<<count;
                }
};

int Counter :: count = 0;              //count initialization with 0

int main()
{
        Counter OB1;
        OB1.Print();

        Counter OB2;
        OB2.Print();

        Counter OB3;
        OB3.Print();

        return 0;
}
```

Output:

```
Total objects are: 1
Total objects are: 2
Total objects are: 3
```

https://hemanthrajhemu.github.io