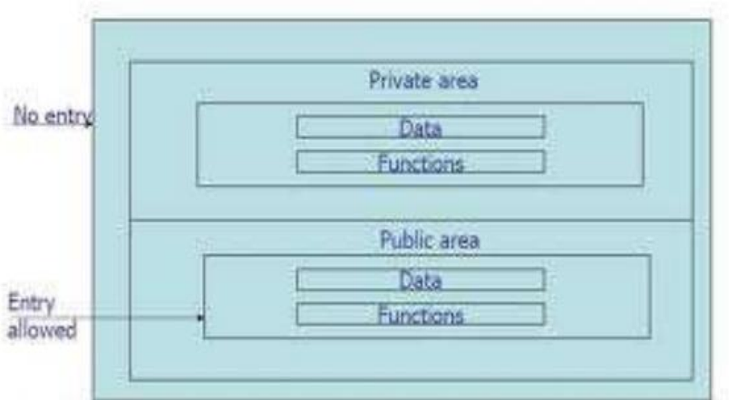
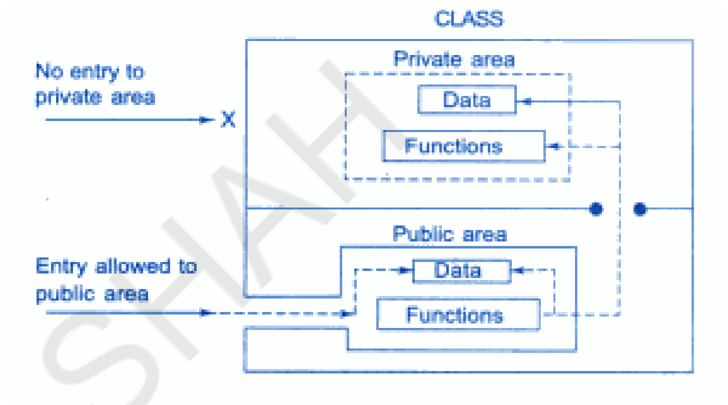
Object Oriented Programming
With
C++

By Shailee Shah
Assistant professor
President Institute of Computer Application

Data hiding in classes





Creation of class and objects

```
class item
{
                    int number;
                    float price;
          public :
                    void getdata(int a, float b);
                    void putdata(int a, float b);
};
```

```
item x;
```

```
item x, y, z;
```

```
Class item
{
.....
}x, y, z;
```

Defining Member function

❑ Two way to define member function :

1. Outside the class definition.

2. Inside the class definition.

❑Outside the class definition:

<u>Syntax :</u>

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

<u>Example :</u>

```
Void item :: getdata (int a, float b)
{
        number = a;
        price = b;
}
```

```
class item
{
                int number;               //private by default
                float price;              //private by default
      public :
                 //function declaration
                void getdata(int a, float b) ;
                void putdata();
};
// function definition
void  item :: getdata (int a, float b)
                {
                                number = a;
                                price = b;
                }
 void  item :: putdata()
                {
                                cout << “No = “ << number;
                                cout << “Price = “ << price;
                }
};
```

```
int main()

{

      item x;

      cout<< “object is X ”<<“\n”;

      // call member function

      x.getdata(10,500);

      x.putdata();

}
```

Output:

```
No =10
Price= 500
```

❑Inside the class definition:

```
class item
{
                int number;
                float price;
        public :
                void getdata(int a, float b)
                {
                        number = a;
                        price = b;
                }
                void putdata()
                {
                        cout << “No = “ << number;
                        cout << “Price = “ << price;
                }
};
```

```
class item
{
            int number;            //private by default
            float price;            //private by default
    public :
            void getdata(int a, float b)
            {
                        number = a;
                        price = b;
            }
            void putdata()
            {
                        cout << “No = “ << number;
                        cout << “Price = “ << price;
            }
};
```

```
int main()

{

    item x;

    cout<< “object is X ”<<“\n”;

    // call member function

    x.getdata(10,500);

    x.putdata();

}
```

Output:

```
No =10
Price= 500
```

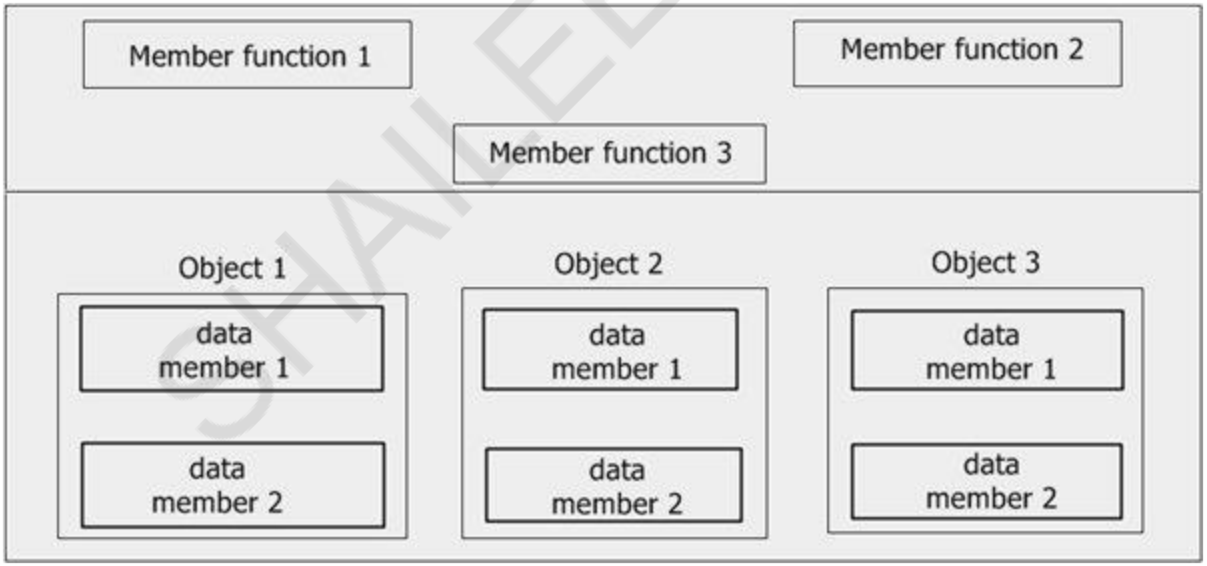
Accessing Class Member

- The main() cannot contain statements that access number and price directly.

```
// not valid
x.no = 10;
x.price = 75.50;

// valid
x.gettdata(10, 75.50);
x.putdata();
```

Memory Allocation for Objects

- The member functions are created and placed in the memory space only once when they are defined as a part of a class specifications.
- Since all the objects belonging to that class use the same member function, no separate space 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.



Static Data Members

- ❑ A data member of a class can be qualified as static.
- ❑ The properties of a static member variable are similar to that of a C’ static variable.

Characteristics of Static Data Member

❑ It is initialized to zero when the first object of it’s class is created. No other initialization is permitted.

❑ Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

❑It is visible only within the class, but its lifetime is the entire program.

Static Data Members

```
#include<iostream.h>
#include<conio.h>
class item{
        static int cnt;
        int number;
   public :
        void getdata(int a)
        {
              number = a;
              cnt++;//1//2//3
        }
        void getcount()
        {
              cout<<"count : "<<cnt<<endl;
        }
};
int item :: cnt;
```

```
void main()
{
   clrscr();

   item a, b, c;

   cout<<"Before\n";

   a.getcount();
   b.getcount();
   c.getcount();

   a.getdata(100);
   b.getdata(200);
   c.getdata(300);

   cout<<"After\n";

   a.getcount();
   b.getcount();
   c.getcount();
   getch();
}
```

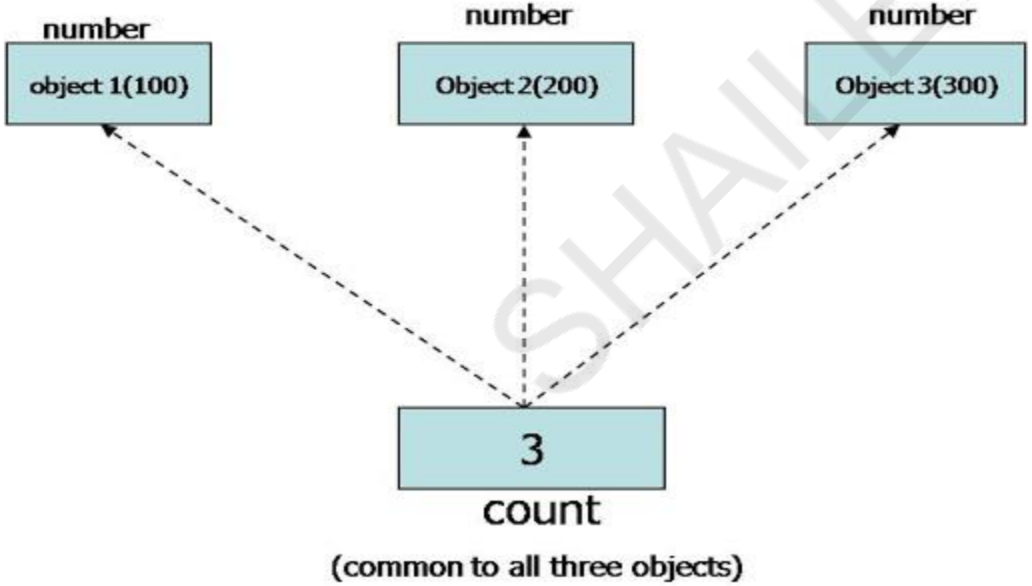
Note : The type and scope of each static member variable must be defined outside the class definition.

data-type class-name :: var-name **(definition of static data member)**

Static Data Members

Output:

```
count : 0
count : 0
count : 0

count : 3
count : 3
count : 3
```



Static Member Function

- Like static member variable, we can also have static member functions.
- Properties :
  - A static function can have access to only other static members (fun. Or var.) declared in the same class.
  - Instead of its objects, A static member function can be called using the class name.

    class-name :: function-name;

Static Members function

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

class item
   {
         static int cnt;
         int number;

   public :

   void getdata(int a)
         {
               number = a;
               cnt++;
         }
         static void getcount()
         {
               cout<<"count : "<<cnt<<endl;
         }
};
```

```
int item :: cnt;

void main()
{
   clrscr();

   item a, b, c;

   cout<<"Before\n";
   a.getcount();
   b.getcount();
   c.getcount();

   a.getdata(10);
   b.getdata(20);
   c.getdata(30);

   cout<<"After\n";

   item::getcount();
   getch();
}
```

Static Members function

Output:

```
count : 0
count : 0
count : 0

count : 3
```

Arrays of Objects

- We know that an array can be of any data type including struct.
- Similarly, we can also have arrays of variables that are of the type class.

```
class employee
    {
            char name[30];
            floar age;
        public :
            void getdata();
            void putdata();
        };
```

- The identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employees.

```
employee manager[5];
employee supervisor[10];
Employee worker[45];
```

- Since an array of objects behaves like any other array, we can use the usual array requests the object manager[i] to invoke the member function putdata().

Arrays of objects

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

class employee
{
        char name[30];
        float age;
   public :
        void getdata();
        void putdata();
};

void employee :: getdata()
{
   cout<<"Enter Name = ";
   cin>>name;
   cout<<"Enter Age = ";
   cin>>age;
}
void employee :: putdata()
{
   cout<<"Name is = "<<name<<endl;
   cout<<"Age is = "<<age<<endl;
}
```

```
const int size=3;

void main()
{
   clrscr();

   employee manager[size];

   for (int i=0; i<size; i++)
   {
        manager[i].getdata();
   }
   for (i=0; i<size; i++)
   {
        cout<<"\nDetails of Manager\n";
        manager[i].putdata();
   }
   getch();
}
```

Objects as Function arguments

- ❑ Like any other data type, an object may be used as a function argument.
- ❑ This can be done in two ways :
  - ➢ A copy of the entire object is passed to the function.
  - ➢ Only the address of the object is transferred to the function.

1. **A copy of the entire object is passed to the function:**

- ❑ It is known as *pass-by-value*.
- ❑ Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

```
void sum(time t1, time t2);
```

2. **Only the address of the object is transferred to the function:**

- ❑ It is known as pass-by-reference.
- ❑ When an address of the object is passed, the called function works directly on the actual object used in the call.

```
void sum(time & t1 , time & t2);
```

- This means that any changes made to the object inside the function will reflect in the actual objects.
- The pass-by-reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Pass-by-value

```
class complex
   {
          int real,img;

          public :

          void getdata(int a, int b)
          {
                 real=a;
                 img=b;
          }
          void putdata()
          {
                 cout<<“real number =“<<real<<endl;
                 cout<<“img number =“<<img<<endl;
          }

          void sum(complex c1,complex c2)
          {
                 real=c1.real+c2.real;
                 img=c1.img+c2.img;
          }
};
```

```
void main()
   {
        clrscr();

        complex c1,c2,c3;
        c1.getdata(2,5);
        c2.gedata(3,6);

        c3.sum(c1,c2);                    // c3=c1+c2
        cout<<“c1 = “<<endl;
        c1.putdata();

        cout<<“c2 = “<<endl;
        c2.putdata();

        cout<<“c3 = ";
        c3.putdata();
        getch();
}
```

```
output:
C1=
real number =2
Img number=5

C2=
real number =3
Img number=6

C3=
real number =5
Img number=11
```

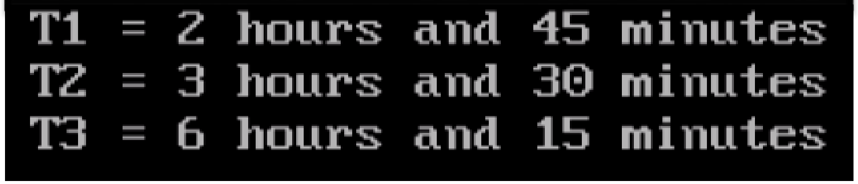
Pass-by-value

```
class time
   {
         int hours,minutes;

         public :

         void gettime(int h, int m)
         {
                hours = h;
                minutes = m;
         }
         void puttime()
         {
                cout<<hours<<" hours and ";
                cout<<minutes<<" minutes \n";
         }
```

```
void sum(time t1, time t2)
          {
                 minutes = t1.minutes + t2.minutes;
                 hours = minutes/60;
                 minutes = minutes%60;
                 hours = hours + t1.hours + t2.hours;
          }
};
```

```
void main()
   {
          clrscr();

          time T1, T2, T3;
          T1.gettime(2,45);
          T2.gettime(3,30);

          T3.sum(T1, T2);                    // T3 = T1 + T2
          cout<<"T1 = ";
          T1.puttime();

          cout<<"T2 = ";
          T2.puttime();

          cout<<"T3 = ";
          T3.puttime();
          getch();
}
```

Output:



```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

Friendly Function

- Normally a non-member function cannot have an access to the private data of a class.
- There could be a situation where we would like two classes to share a particular function.
- For example, Two classes **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the object of both these classes.
- In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes.
- To make an outside function "**friendly**" to a class, we have to simply declare this function as a ***friend*** of the class.

```
class ABC{
                    . . . . .
                    . . . . .
          public
                    . . . . .
                    . . . . .
                    friend void xyz();
}
```

Friendly Function

- ❑ The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function.
- ❑ The function definition does not use wither the keyword **friend** or **the scope operator : :**
- ❑ The function that are declared with keyword **friend** are known as **friend function.**
- ❑ A function can be declared as a friend in any number of classes.
- ❑ A friend function, although not a member function, has full access right to the private member of the class.

Characteristics of Friendly Function

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member function, it cannot access the member names directly and has to use an object name and do membership operator with each member name. (ex. A.c)
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

Friend Function With One Class

```
#include<iostream.h>
#include<conio.h>
class sample
        {
                int a,b;
        public:
                void setvalue()
                {
                a=25; b=40;
                }

        friend float mean(sample s);

        };
float mean(sample s)
        {

        return float(s.a+s.b)/2.0;

        }
```

```
void main()
{
        clrscr();
        sample x;
        x.setvalue();
        cout<<"Mean is = "<<mean(x);
        getch();
}
```

Output:

```
Mean is=32.5
```

Friend Function With Two Class

```
#include<iostream.h>
#include<conio.h>
class ABC;
class XYZ
        {
                int x;
        public:
                void setvalue(int i)
                {
                        x=i;
                }
                friend void max(XYZ, ABC);
        };

class ABC
        {
                int a;
        public:
                void setvalue(int i)
                {
                a=i;
                }
                friend void max(XYZ, ABC);
        };
```

```
void max(XYZ m, ABC n)
        {
        if(m.x >= n.a)
                cout<<m.x;
        else
                cout<<n.a;
        }

void main()
        {
                clrscr();

                ABC p;
                XYZ q;

                p.setvalue(10);

                q.setvalue(20);

                max(q,p);

                getch();
}
```

Returning Objects

❑ A function not only receive objects as arguments but also can return them.

```
#include<iostream.h>
#include<conio.h>
class complex
{
        float x,y;
     public:

        void input(float r, float i)
        {
               x=r;
               y=i;
        }

        complex sum(complex, complex);
        void show(complex);
};
complex sum(complex c1, complex c2)
{
     complex c3;
     c3.x = c1.x + c2.x;
     c3.y = c1.y + c2.y;
     return(c3);
}
```

```
void complex :: show(complex c)
{
          cout << c.x << " : " << c.y << "\n";
}
void main()
{
          clrscr();

          complex A,B,C;

          A.input(3.1, 5.65);
          B.input(2.75, 1.2);
          C = sum(A, B);

          cout<<"A = ";
          A.show(A);

          cout<<"B = ";
          B.show(B);

          cout<<"C = ";
          C.show(C);
          getch();
}
```

const Member Function

- If a member function does not alter any data in the class, then we may declare it as a **const** member function.

  double get_balance() const;

- The qualifier **const** is appended to the function prototypes (in both declaration and definition).

- The compiler will generate an error message if such function try to alter the data value.

Dynamic Memory Management

Memory Management Operators

- C uses **malloc()** and **calloc()** function to allocate memory and **free()** to free memory dynamically at run time.
- We use dynamic allocation techniques when it is not known in advance how much of memory space is needed.
- C++ defines two unary operators :

  new

  delete
- to perform this task in better and easier way.
- An object can be created by using ***new***, and destroyed by using ***delete***, as an when required.
- Lifetime of an object is directly under our control and is unrelated to the block structure of the program.
- new:

  Syntax : *pointer-variable = new data-type;*

  Example : int *p = new int;
  *p = 25;
  float *q = new float(8.5);

❑ For array

Syntax : *pointer-variable = new data-type[size];*

Example : int *p = new int[10];

❑ delete :

Syntax : *delete [size] pointer-variable;*

Example : delete [] p;

Advantages of ***new*** over malloc()

❑ It automatically computes the size of the data object. We need not use the operator sizeof.

❑ It automatically returns the correct pointer type, so that there is no need to use a type cast.

❑ It is possible to initialize the object while creating the memory space.

❑ Like any other operator, ***new*** and ***delete*** can be overloaded.

Thank you

☺