Chapter 3:
Classes and Objects

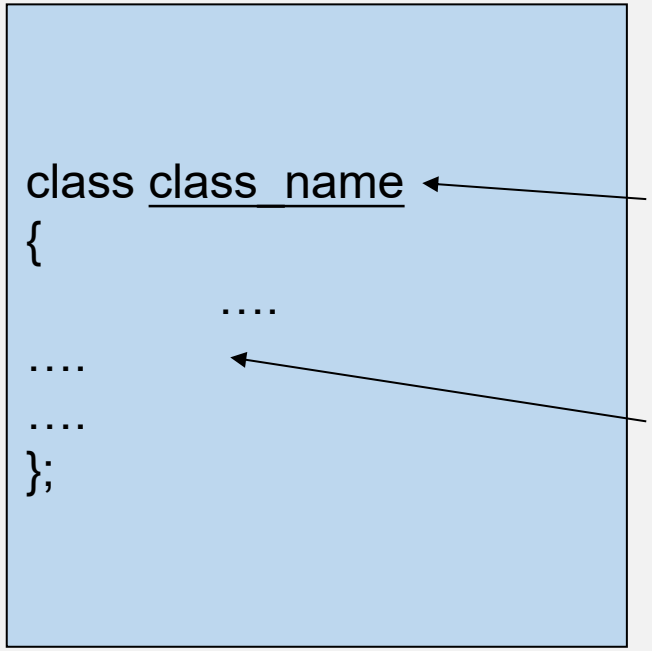
BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

Classes

- Classes are templates and specification of the object
- Also a user defined data type
- An encapsulation technique that binds data and its associated function together in single unit
- It is an ADT because it represents essential features without including background detail or explanation

Classes

- A class definition begins with the keyword ***class***.
- The body of the class is contained within a set of braces, **{ } ;** (notice the semi-colon).

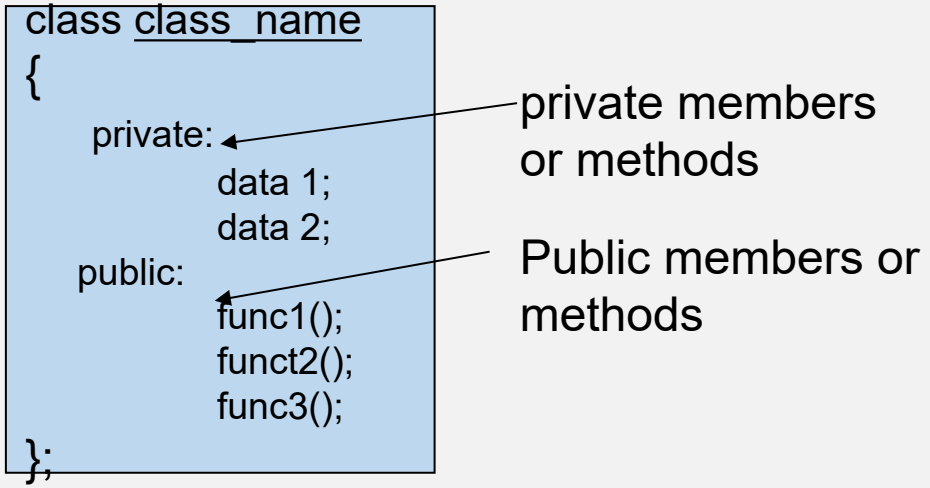


Any valid identifier

Class body (data members+ **member functions**)

Access Specifiers

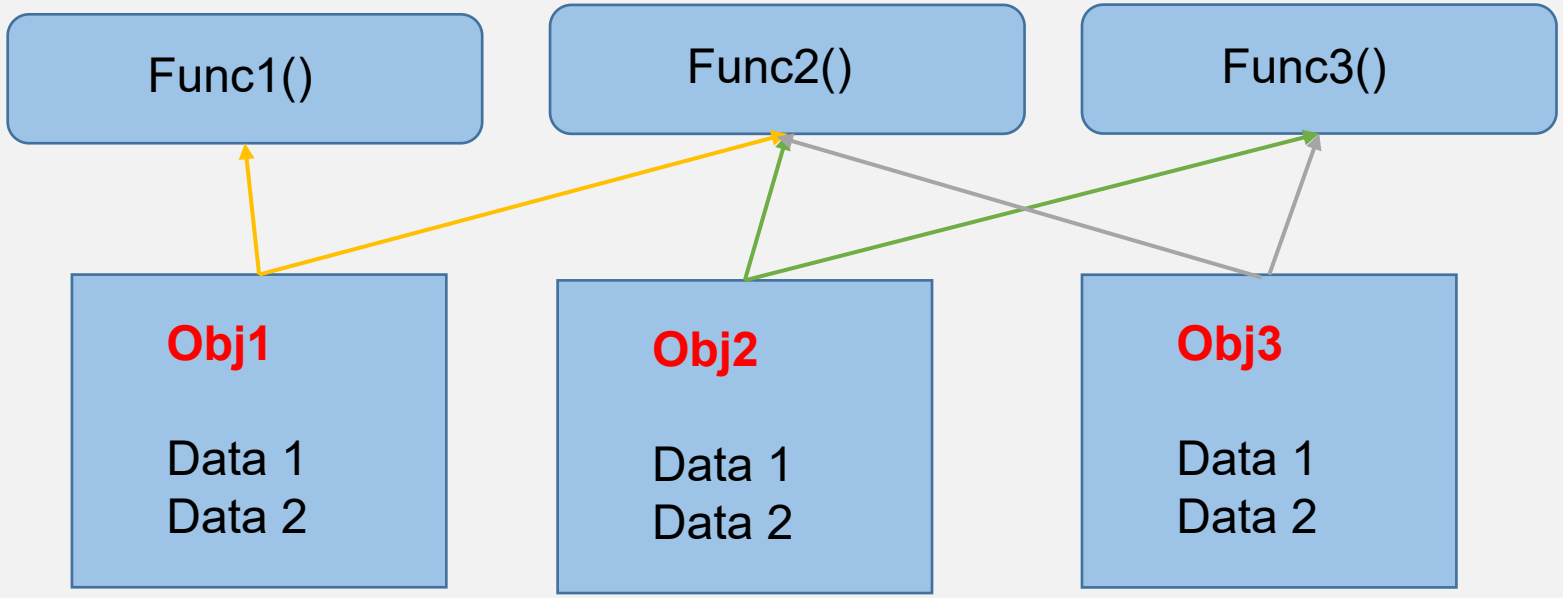
- They determine the visibility of the data member and member functions defined in the class
- One access specifier can occur more than once in a class definition
- Three types:
  - **Private**:members cannot be accessed from outside the class
  - **Public**: members are accessible from outside class
  - **Protected**: members cannot be accessed from outside the class, however, they can be accessed in inherited classes



- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.

Objects

- Objects are runtime variables or the instances of class
- An object has the same relationship to a class that a variable has to a data type
- To declare object,
  - Class_name object_name;



Accessing members of class

- The class members can be accessed by using **dot (.) operator**.
- The dot (.) operator is called member access operator.
- Members are accessed as,
  - Object_name.member_name;
    - Obj1.data_mem; // accessing public data
    - Obj1.func1(); // accessing public function

Defining member functions

- The member functions can be defined
  - Inside the class
  - Outside the class
- The member functions are inline (by default) if they are defined inside the class. Otherwise, they have to be defined as “inline” explicitly if they are defined outside the class
- One member function can invoke other member function as in normal functions
- Member functions can also be overloaded and take default arguments.

Defining member functions

- 1. Defining inside class

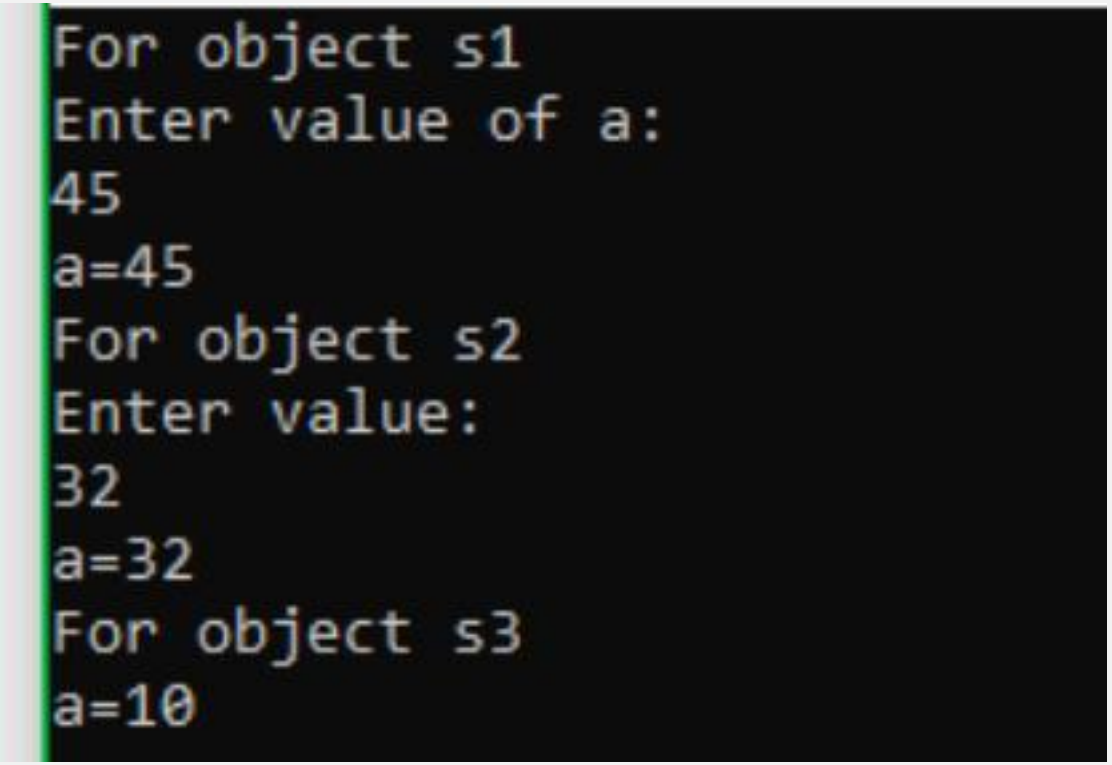
```
class student
    {
        int roll;
        char name[20];
    public:
            void getdata()
            {
            cout<<“Enter roll and name”;
            cin>>roll>>name;
                    }
    };
```

- 2. Defining outside class

```
class student
    {
        int roll;
        char name[20];
    public:
            void getdata();
    };

    void student::getdata()
    {
            cout<<“Enter roll and name”;
            cin>>roll>>name;
    }
```

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

class alpha
{
private:
    int a;
public:
    void setdata()
    {
        a=10;
    }

    void setdata(int x)
    {
        a=x;
    }
    void getdata()
    {
       cout<<"Enter value of a:"<<endl;
        cin>>a;
    }

    void writedata()
    {
        getdata();
        cout<<"a="<<a<<endl;
    }
    void display();
    void show ();
};

void alpha::display()
{
cout<<"a="<<a<<endl;
}

inline void alpha::show()
{
      int i;
    cout<<"Enter value:"<<endl;
    cin>>i;
    setdata(i);
    cout<<"a="<<a<<endl;
}
```

```
int main()
{
   alpha s1, s2, s3;
    cout<<"For object s1"<<endl;
    s1.writedata();
    cout<<"For object s2"<<endl;
    s2.show();
    cout<<"For object s3"<<endl;
    s3.setdata();
    s3.display();
}
```



Constructors

- Special member function that is executed automatically during object creation
- Used to provide value(initialization) to the data member
- No return type(not even void)
- Takes name same as class name
- Public function and can be overloaded
- Three types:
  - Default constructors
  - Parameterized constructor
  - Copy constructor

Constructors- Default Constructors

- Default constructor is a constructor without argument

```
class alpha
{       int a;                                  int main()
        public:                                 {
        alpha()                                 alpha s1;
        {       a=10; }                         }
        // using initializer list
        // alpha() : a(10) {  }
};
```

- If no constructor is defined in the class then the compiler automatically creates one for the program which doesn't take any parameters

Constructors- Parameterized Constructors

- The constructor that can take arguments is called parameterized constructor.
- If we often need to initialize the various data elements of the different object with different values when they are created, then it can be achieved by passing the arguments to the constructor functions when the object is created.

```
class alpha
        {       int a;
                public:
                alpha(int x)
                {       a=x;    }
                // using initializer list
                // alpha(int x) : a(x) {  }
        };
```

```
int main()
{
alpha s1(25);
}
```

Constructors- Copy Constructors

- It creates a new object as a copy of an existing object.
- For the classes which do not have a copy constructor defined by the user, compiler itself creates a copy constructor for each class known as default copy constructor.

```
class alpha
    {       int a;
            public:
            alpha(alpha &x)
            {       a=x.a;   }
            // using initializer list
            // alpha(alpha &x) : a(x.a) {  }
            alpha(int x): a(x) { }
    };
```

```
int main()
{
alpha s1(25);
alpha s2(s1);
alpha s3=s1;
}
```

Constructors : Points to be noted

- After defining parameterized constructor, it is mandatory to define default constructor because system doesn’t generate default constructor.
- For the ‘const’ and ‘reference’ members, we must always define constructor
- For parameterized constructor,
  - alpha a(5); //implicit call
  - alpha a=alpha(5); //explicit call, where alpha(5) is temporary un-named object

Destructors

- A special member function that is executed automatically when an object is destroyed that has been created by the constructor.
- Used to de-allocate the memory that has been allocated for the object by the constructor.
- Name preceded by tilde(~) sign
- No argument and no return type,
- Only one destructor in a class whereas constructors can be overloaded
- Can be virtual, but constructors cannot be virtual

Constructor Overloading

- Like other member functions, constructor can also be overloaded
- Constructor can also take default argument

```
class alpha
    {       int a;
            public:
            alpha(int x): a(x) { }
             alpha() : a(10) {  }
             alpha(alpha &x) : a(x.a) {  }
              ~alpha() {  }
                    void  display()
                   {        cout<<a<<endl;
                   }
    };
```

```
int main()
{
alpha s1 , s2(20) , s3(s2);
s1.display();
s2.display();
s3.display();
}
```

Constructor and Destructor- Invocation Order

- The constructors are invoked in sequence as the objects are created
- Whereas destructors are invoked in reverse order
- In above program, constructors are invoked as,

  s1.alpha() → s2.alpha(20) → s3.alpha(s2)
- The destructors are invoked as,

  s3.~alpha() → s2.~alpha() →s1.~alpha()

```
class alpha
        {       int obj_no;
    public:
                alpha(int x ): obj_no(x)
                {       cout<<"object "<<obj_no<<" created"<<endl;  }
                ~alpha()
                {       cout<<"object "<<obj_no<<" destroyed"<<endl;       }
        };
main()
        {
           alpha s1(1),s2(2),s3(3);
           {
              alpha s4(4);
           }
           alpha s5(5);
        }
```

```
object 1 created
object 2 created
object 3 created
object 4 created
object 4 destroyed
object 5 created
object 5 destroyed
object 3 destroyed
object 2 destroyed
object 1 destroyed
```

Dynamic Constructor

- When allocation of memory is done dynamically using dynamic memory allocator **new** in a **constructor**, it is known as **dynamic constructor**.

```
class alpha
        {
                int len;
                char *str;
        public:
                alpha()
                {    str=new char[1];
                   strcpy(str," ");
                }
                alpha(char *s)
                {    len=strlen(s);
                   str=new char[len+1];
                   strcpy(str,s);
                }
                ~alpha()
                {
                     delete [ ] str;
                }
        void display()
                {
                   cout<<str;
                }
};

                main()
                {
                   alpha s1("Pulchowk");
                   s1.display();

                }
```

Object as argument

- Like any other data type, an object may be used as A function argument. This can Done in two ways:
- A copy of the entire object is passed to the function. **(Pass by Value).**
- Only the address of the object is transferred to the function. **(Pass by Reference).**

```
class comp
{
    int real, imag;
    public:
        comp():real(0),imag(0){ }
        comp(int r, int i):real(r), imag(i){ }
        void add(comp,comp);
        void display()
        {
            cout<<real<<"+"<<imag<<"i"<<endl;
        }
};
void comp::add(comp c1, comp c2)
{
    real=c1.real + c2.real;
    imag=c1.imag + c2.imag;
}
```

```
main()
{
    comp n1(10,20),n2(30,40), n3;
    n3.add(n1,n2);
    cout<<"First number:";
    n1.display();
    cout<<"Second number:";
    n2.display();
    cout<<"Final number after addition:";
    n3.display();
}
```

```
First number:10+20i
Second number:30+40i
Final number after addition:40+60i
```

Object as return type

- A function can also return objects either by value or by reference.
- When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.
- In the case of returning an object by reference, no new object is created, rather a reference to the original object in the called function is returned to the calling function

```
class comp
{
   int real, imag;
   public:
      comp():real(0),imag(0){}
      comp(int r, int i):real(r), imag(i){}
      comp add(comp);
      void display()
      {

cout<<real<<"+"<<imag<<"i"<<endl;
      }
};
```

```
comp comp::add(comp c2)
{
    comp temp;
    temp.real = real + c2.real;
    temp.imag = imag + c2.imag;
    return temp;
}

main()
{
   comp n1(10,20),n2(30,40), n3;
   n3=n1.add(n2);
   cout<<"First number:";
   n1.display();
   cout<<"Second number:";
   n2.display();
   cout<<"Final number after addition:";
   n3.display();
}
```

Constant object and constant member function

- Constant object
  - Whose data member values cannot be changed
  - Can invoke constant member function only
  - Declared as,
    - const class_name object_name;

Constant object and constant member function

- Constant member function
  - Function in which data members remain constant
  - Uses ‘const’ as suffix in function declaration/definition
    - Return_type function_name(arg/s) const;

```
class alpha
        {
              int a;
        public:
                    alpha() : a(10) {  }
                    void display()
                     {
                        a+=10;
                        cout<<a<<endl;
                     }
                     void show()const
                     {
                     // a+=10; not permitted
                    cout<<a<<endl;
                     }
        };
```

```
main()
        {
           alpha s1 ;
           const alpha s2;
           s1.display();
           //s2.display(); cannot invoke non-const func
           s1.show();
           s2.show();
        }
```

Static data member

- 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 declared as,
  - static data_type member;
- Accessible within class and have lifetime of entire program.
- It is initialized outside the class because its scope is not limited to class but entire program.
  - Data_type class_name :: member =value;
  - OR
  - Data_type class_name :: member; // takes zero value by default

```
class item
{
static int count; //count is static
int number;
public:
void getdata()
{
    number=count++;
}
void get_count(void)
{
    cout<<”count:”;
    cout<<count<<endl;
}
};
```

```
int item :: count ; //count defined
int main( )
{
item a,b,c;
a.get_count( );
b.get_count( );
c.get_count( );

a.getdata( );
b.getdata( );
c.getdata( );

cout«"after reading data : "«endl;
a.get_count( );
b.get_count( );
c.get_count( );
return(0);
}
```

The output would be
count:0
count:0
count:0
After reading data
count: 3
count:3
count:3

Static member function

- Can access static data members only
- To invoke static member function, we do not need to access through object using member access operator(.) but rather uses class name with scope resolution operator because it refers to entire class rather than specific object
  - Class_name:: static_function();

```
class test
{
int code;
static int count; // static member variable
public:
test()
{
code=++count;
}
void showcode()
{
cout<<"object member : "<<code<<endl;
}
~test()
{
cout<<"Object "<<code<<" destroyed"<<endl;
count--;
}
static void showcount()
{ cout<<"count="<<count<<endl; }
};

int test:: count;

int main()
{
test t1,t2;
test :: showcount ( );
test t3;
test:: showcount( );

t1.showcode( ) ;
t2.showcode( );
t3.showcode( );
test:: showcount( );
return(0);
}
```

```
count=2
count=3
object member : 1
object member : 2
object member : 3
count=3
Object 3 destroyed
Object 2 destroyed
Object 1 destroyed
```

Pointer to object and member access

- As in basic data types and array, pointers can also point to object as well
  - Class_name *object;
- Uses indirection operator(->) for accessing members as,
  - object->member; // (*object).member

```
class alpha
        {       public:
                 void showaddress()
                 {    cout<<this<<endl;  }
        };
main()
        {   alpha s;
            alpha *sp =&s;  // or, alpha *sp = new alpha;
            sp->showaddress(); // or ,   (*sp).showaddress();
}
```

DMA for object and object array

- **For single object,**

```
main()
        {   alpha *sp1;
            sp1=new alpha;
            sp1->showaddress();

           alpha *sp2=new alpha;
            sp2->showaddress();

 }
```

- **For array of objects,**

```
main()
          {   int n;
              cin>>n;
              alpha *sp;
            sp=new alpha [n];
            for(int i=0;i<n;i++)
         {
           sp[i]=new alpha;
              sp[i]->showaddress();
            }
 }
```

‘this’ pointer

- When a member function(non static) is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**.
- ‘this’ pointer contains the address of that object

```
class alpha
        {       public:
                 void showaddress()
                 {    cout<<this<<endl;  }
        };
        main()
        {   alpha s1 ;
            s1.showaddress();
        }
```

‘this’ pointer

- The private variable ‘a’ can be used directly inside a member function, like a=123;
- We can also use the following statement to do the same job.
  - this → a = 123

- Also used to remove ambiguity among data members

```
class alpha
{          int a;
public:
void setdata(int a)
 {     this-> a = a;
      }
};
```

‘this’ pointer

- It acts as implicit argument to all member function. Hence can be used to return object.

```
comp comp::add( comp c2)
{
   this ->real += c2.real;     //  real += c2.real;
   this->imag += c2.imag;  // imag += c2.real;

   return *this;
}
```

Which is invoked as c3=c1.add(c2)

Friend functions

- Non-member function defined outside the class but can access all data members of class
- Prototype appears within the class(either in private or public section)
- Prototype is preceded by keyword “friend”
- takes object as argument (or create object) as it cannot access private data members directly
- Invoked as normal function without object because it is non-member function and does not belong to a class

Friend functions

- 1. It can access private members of a class

```
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);
}

main ( )
{
sample x;
x . setvalue( );
cout<<"mean value="  <<mean(x)<<endl;
}
```

Friend functions

- 2. It can act as bridge between two or more classes

```
class abc; //forward declaration

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;
}

int main( )
{
abc j;
j . setvalue( 10);
xyz s;
s.setvalue(20);
max( s , j );
}
```

Friend functions

- Member function of one class can be friend of another.

```
class B;
class A
{
public:
    void show(B&);
};

class B
{    int b;
public:
    B():b(20){ }
    friend void A::show(B&);
};
```

```
void A::show(B &x)
{
    cout<<x.b;
}

main()
{
    A m;
    B n;
    m.show(n);
}
```

Friend Class

- A friend class can access all the private and protected data members of the class to which it is friendly.
- Member function of the friendly class becomes the friend function to the class to which it is friendly.
- Friend class is not mutual. That is, if the first class becomes friend to second class, then it is not necessary that second class is also friend to first class.

Friend Class

```
class B;
class A
{    int a;
public:
    A():a(10){ }
    void show(B&);
};

class B
{    int b;
public:
    B():b(20){}
    friend class A;
};
```

```
void A::show(B &x)
{
    cout<<a<<x.b;
}


main()
{
    A m;
    B n;
    m.show(n);
}
```

Note:

1. friend class A can access private member of B
2. Member function of class A becomes friend of B which can access data of B.