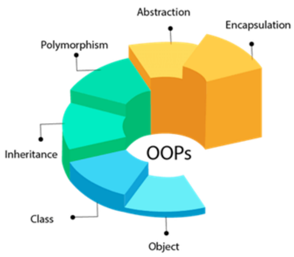
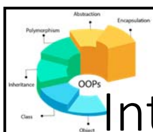


Objects and Classes

Chapter 3

Object Oriented Programming

By DSBaral



Introduction

- Classes forms basis of object oriented programming.
- Class provide a tool for creating new data types called as abstract data type that can be used as conveniently as the built in types.
- A class is actually a user defined data type like structure in C.
- A class encloses both data and functions that operate on the data into a single unit.
- Member functions define the various operations on the data members of a class and provide access to the class.
- Objects are the variables or instances of classes.

Class Definition

- A class definition is syntactically similar to a structure definition.
- Classes contain data as well as functions in a single unit.
- The general format of declaring class is as follows:

Access specifiers

```
class class_name
{
    private:
          data_type data1;
          //......
    public:
          return_type function_name(data_type arg_list);
          //......
};
```


Class Definition (Contd…)

- The class members (data and functions) are normally grouped into sections `private`, `public` or `protected`, called as access specifiers or visibility levels.
- The `private` members are not visible from outside the class where as `public` members are visible anywhere from the program.
- There is also another access specifier `protected` which will be discussed later.
- In absence of access specifiers members will be private by default.

Object Declaration and Member Access

- The objects of the class can be declared similar to the variable declaration as
  `class_name object_name;`
- The data members of the object are accessed using the member access operator (`.`) as
  `object_name.data_member_name;`
- The function members of the object can be accessed as
  `object_name.member_function_name(arguments…);`

Simple Class Example

- Lets see an example of a program with simple class

```
class test
{
private:
    int data1;
    int data2;
public:
    void setdata(int d1, int d2) {
        data1=d1;  //functions defined inside the class
        data2=d2;  //are treated inline
    }
    void showdata() {
        cout<<"Data member 1= "<<data1<<endl;
```

Simple Class Example (Contd…)

```
        cout<<"Data member 2= "<<data2<<endl;
    }
};
int main()
{
    test t1,t2; //t1.data1 -> error
    t1.setdata(101,102);
    t2.setdata(201,202);
    cout<<"\nObject 1 Data:"<<endl;
    t1.showdata();
    cout<<"\nObject 2 Data:"<<endl;
    t2.showdata();
    return 0;
}
```


Defining Functions Outside the Class

- The member function of a class can also be defined outside the class definition

```
class   class_name
{
    //......
public:
    return_type member_function(params..);
    //.....
};
```

return type

owner Class

function name

parameters

```
return_type class_name::member_function(params..)
{
    //function body
}
```

Defining Functions Outside the Class (Contd…)

- In the above example member functions are defined inside the class but they can also be defined outside the class as

```
class test
{
private:
    int data1;
    int data2;
public:
    void setdata(int d1, int d2);
    void showdata();
};
```


Defining Functions Outside the Class (Contd…)

owner class   member function name   parameters

return type

```
void test::setdata(int d1, int d2)
{
    data1=d1;  //Functions defined outside the class
    data2=d2;  //definition are not inline
}
void test::showdata()
{
    cout<<"Data member 1= "<<data1<<endl;
    cout<<"Data member 2= "<<data2<<endl;
}
```

Defining Functions Outside the Class (Contd…)

- The functions defined outside the class definition are not inline.
- To make such functions inline we have to prefix the keyword `inline` in front of the function header as

```
inline void test::showdata()
{
    cout<<"data1= "<<data1<<endl;
    cout<<"data2= "<<data2<<endl;
}
```


A Physical Object Example

- A more practical example that represent physical object is

```
class product
{
private:
    int productid;
    char name[15];
    float cost;
public:
    void setdata(int pid,char pname[],float cst)
    {
        productid=pid;
        strcpy(name,pname); //name=pname;
        cost=cst;
    }
```

A Physical Object Example (Contd…)

```
        void showdata()
        {   cout<<"Product ID: "<<productid<<endl;
            cout<<"Name: "<<name<<endl;
            cout<<"Cost: "<<cost<<endl;
        }
    };
    int main()
    {   product p1,p2;
        p1.setdata(944,"CD-ROM ", 1500.00);
        p2.setdata(945,"Pen Drive", 1000.00);
        p1.showdata();
        p2.showdata();
        return 0;
    }
```


A Mathematical Data as Object Example

- An example of a program that uses mathematical data as object.

```
class  complex
{
private:
    float real;
    float imag;
public:
    void readvalue()
    {   cout<<"Enter Real part: "; cin>>real;
        cout<<"Enter Imaginary part: "; cin>>imag;
    }
```

A Mathematical Data as Object Example (Contd...)

```
        void showvalue()
        {   cout<<"("<<real<<","<<imag<<")"; }
        void cadd(complex cn1, complex cn2)
        {   real=cn1.real+cn2.real;
            imag=cn1.imag+cn2.imag;
        }
    };
    int main()
    {   complex c1,c2,c3;
        cout<<"Enter first complex number: "<<endl;
        c1.readvalue();
```


A Mathematical Data as Object Example (Contd...)

```
        cout<<"\nEnter second complex number: "<<endl;
        c2.readvalue();
        c1.showvalue();
        cout<<" + ";
        c2.showvalue();
        c3.cadd(c1,c2);
        cout<<" = ";
        c3.showvalue();
        return 0;
    }
```

Returning Objects from Functions

- An example that returns objects from function is as follows

```
class  complex
{
private:
    float real, imag;
public:
    void setvalue(float re, float im)
    {   real=re; imag=im; }
    void showvalue()
    {   cout<<"("<<real<<","<<imag<<")";
    }
```

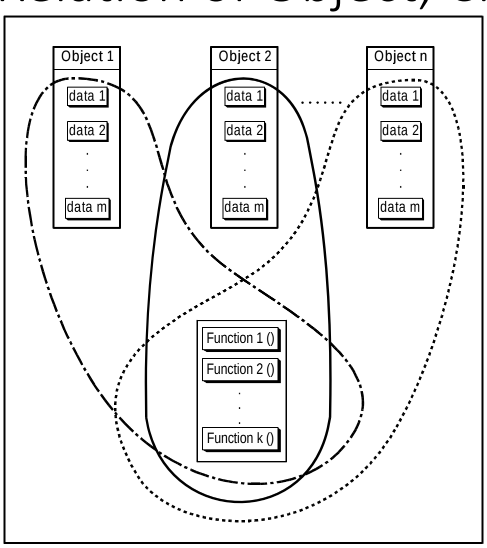

Returning Objects from Functions (Contd...)

```
    complex cadd(complex cn)
    {   complex res;
        res.real=real+cn.real;
        res.imag=imag+cn.imag;
        return res;
    }
};
int main()
{
    complex c1,c2,c3;
```

Returning Objects from Functions (Contd…)

```
        c1.setvalue(2.4,7.5);
        c2.setvalue(3.2,5.6);
        c3=c1.cadd(c2); //c3=c1+c2;
        cout<<"\nComplex num1="; c1.showvalue();
        cout<<"\nComplex num2="; c2.showvalue();
        cout<<"\nComplex num3="; c3.showvalue();
        return 0;
    }
```


Relation of Object, Class and Memory



- The creation of class develops a template for object creation but does not allocate the memory.
- It is known that objects contain data and function together so when objects are created memory space is allocated for each member of objects.
- This mental model helps in programming which assist in understanding abstraction mechanism.
- However in reality memory is allocated only for data members but not for function members

Constructor

- The class feature in C++ provides special member functions that are automatically invoked when objects are created.
- So constructors can be used for the necessary initialization or startup actions for the objects.
- A constructor
  - Is member function that has same name as the class
  - Does not have return type
  - Invoked automatically during object creation
  - Can be overloaded for different ways of startup/initialization
  - Cannot be invoked explicitly other than creating objects

Constructor (Contd…)

- The constructor is made as follows

```
class class_name
{
private:
    //…
public:
    class_name(); //constructor function
    //…
};
```

Constructor (Contd...)

- The constructors are created as follows

```
class counter
{
private:
    unsigned count;
public:
    counter() {count=0;}  //default constructor
    counter(unsigned n) {count=n;}
                          //parameterized constructor
    void inc(){count++;}
    int val(){return count;}
};
```

constructor overloading

Constructor (Contd...)

- The constructors are used as follows

```
int main()
{
   counter c1,c2(5);  //c1.count=0, c2.count=5
   cout<<"\nCounter values before increment:";
   cout<<"\n  Counter 1="<<c1.val();   //c1=0
   cout<<"\n  Counter 2="<<c2.val();   //c2=5
   c1.inc();
   c1.inc();
   c2.inc();
   cout<<"\nCounter values after increment:";
   cout<<"\n  Counter 1="<<c1.val();   //c1=2
   cout<<"\n  Counter 2="<<c2.val();   //c2=6
}
```

Constructor (Contd…)

- The constructors without parameter are called default constructors.
- The constructors with parameter(s) are called parameterized constructors.
- Even when we do not define any constructor in a class a constructor is implicitly defined by the compiler
  - The compiler generated default constructor does nothing except calling default constructor of base class or default constructors of object members.
- The default constructor `counter::counter()` is called when creating object of the class counter without supplying arguments as:

```
counter c1;
```


Constructor (Contd…)

- The parameterized constructor are called when creating objects by passing arguments as:

```
counter c1(2); //int a=5; int a(5); int a=int(5);
```

- The behavior of default constructor can also be performed by a parameterized constructor with the default value as:

```
class counter
{
private:
   unsigned count;
public:
   counter(unsigned n=0){count=n;}
   //......
};
```

This constructor serves as a default and parameterized constructor.

Constructor (Contd…)

- Instead of assigning the data members of a class in the body of the constructor we can also initialize the members through the member initializer list as

```
class counter
{
private:
    unsigned count;
public:
    counter():count(0){}
    counter (unsigned n):count(n){}
   //......
};
```


Constructor (Contd…)

- To initialize multiple members through the member initializer list the members are separated by comma as

```
class cname
{
private:
    int data1;
    int data2;
public:
    cname():data1(0),data2(0){}
    cname (int n1, int n2):data1(n1),data2(n2){}
   //......
};
```

Constructor (Contd…)

- A single parameter constructor is invoked as

```
counter c1=5; //counter c1=counter(5);
```

- For `const` and reference members one must provide the constructor because they must be initialized and the compiler generated default constructor does not work. For example

```
class test
{
   const int n;
   int &r;
};
test t;//error
```


Constructor (Contd…)

- The correct form is:

```
int num=5;
class test
{   const int n;
    int &r;
public:
    test():n(0),r(num){}
    test(int a,int &b):n(a),r(b) {}
};
test t; //ok
test t2(2,num); //ok
```

- Constant and reference members must be initialized through initializer list
- The C++ built in types also have constructs similar to default constructors.

Constructor (Contd…)

- When we declare at least one constructor the compiler will not generate the default constructor.

```
class test
{
private:
   int data;
public:
   test(int n){data=n;}//parameterized constructor
   //......
};
test t1; //error no default constructor
test t2(5) //ok, parameterized constructor is called
```


Constructor (Contd…)

- To be safe we must declare the default constructor if we have parameterized constructor as:

```
class test
{
private:
   int data;
public:
   test(){} //default constructor
   test(int n){data=n;}//parameterized constructor
   //......
};
test t1; //ok, user defined default constructor is called
test t2(5); //ok, parameterized constructor is called
```

Copy Constructor

- Objects can also be initialized by another object of its own type as

```
test t1(5);//invokes one parameter constructor
test t2(t1);//initialize t2 with the value of t1
test t3=t1;// initialize t3 with the value of t1
```

  Here in second and third cases implicitly generated constructor that takes object of its own type.
- The constructor that takes object of its own type as its argument is called *copy constructor*.
- The default behavior of implicitly generated copy constructor is that it copies each member one by one similar to assigning one structure (object) to another.

Copy Constructor (Contd...)

- If the programmer need to do something different than the default behavior of the implicitly generated copy constructor then one can override it.
- The copy constructor is defined as

```
class test
{
private:
   //......
public:
   test(){}
   test(params...){...}
   test(test &t){...}//copy constructor
};
```

Copy Constructor (Contd…)

- The copy constructor is invoked in the following cases:

```
test t1;
test t2(t1);  //copy constructor called
test t3=t1;   //copy constructor invocation
```

- But, the assignment will not call the copy constructor

```
t3=t2;     //assignment, not initialization
test t4=t3;//object creation and initialization
           //copy constructor invocation
```


Copy Constructor (Contd…)

- The class with copy constructor is defined as

```
class counter
{
private:
    unsigned count;
public:
    counter (int n=0) {count=n;}
    counter (counter &c)
    {   count=c.count;
        cout<<"Copy Constructor Invoked";
    }
    void inc(){count++;}
    int val(){return count;}
};
```

Copy Constructor (Contd…)

- This copy constructor can be used as

```
int main()
{
    counter c1(5), counter c4;
    counter c2(c1);  //Copy constructor called
    counter c3=c1;   //Copy constructor invoked
    c4=c1;  //assignment, no copy constructor called
    cout<<"\nCounter values before increment:";
    cout<<"\n  Counter 1="<<c1.val();//c1=5
    cout<<"\n  Counter 2="<<c2.val();//c2=5
    cout<<"\n  Counter 3="<<c3.val();//c3=5
```


Copy Constructor (Contd…)

```
    c2.inc();
    c3.inc();
    c3.inc();
    cout<<"\nCounter values after increment:";
    cout<<"\n  Counter 1="<<c1.val();  //c1=5
    cout<<"\n  Counter 2="<<c2.val();  //c2=6
    cout<<"\n  Counter 3="<<c3.val();  //c3=7
    return 0;
}
```

- ***If a class has reference and `const` members then copy constructor cannot be created and assignment operator cannot be overloaded.***

Destructor

- Similar to constructors, destructors are special function.
- Destructors are invoked when object is being destroyed.
- The destructor's name is same name of the class preceded by a tilde (~) character.
- Destructor does not take any argument and return a value.
- There can be only one destructor in a class.
- It is not necessary to make a destructor if final cleanup is not necessary.
- The implicitly generated destructor does nothing except destroying sub objects or member objects

Destructor (Contd..)

- Destructors are normally used to release memory acquired by the constructor or to perform some other cleanup operation for the object.
- Following is the format for declaring the destructor in a class

```
class Test
{
private:
        //......
public:
        Test(){}//constructor
        ~Test(){}//destructor
 };
```

Destructor (Contd..)

- Let's see the usage of destructor

```
class test
{
private:
    int data;
public:
    test(){data=0;}
    test(int n){data=n;}
    //...
    ~test(){cout<<"\nobject "<<data<<" destroyed";}
};
int main()
{   test c1, c2(5);
    //...
}
```


Destructor (Contd..)

- When dynamic memory allocation is done destructor is used as

```
class test
{
private:
    int *arr;
public:
    test(){}
    test(int n){arr=new int[n];}
    //...
    ~test(){ delete []arr;}
};
int main()
{   test c2(5); //constructor allocates space for 5 ints
                //destructor deallocates allocated space
```

Class and Structure

- In C++, declaring structure is fundamentally the same as declaring a class.
- The structure in C++ can have member function along with data members in a single unit.
- Unlike the structures in C, in C++ they are alternative way of declaring class.
- The only difference between class and structure is that, in class, members are private by default where as in structure members are public by default.

Class and Structure (Contd…)

- When structure is defined as

```
struct s
{
   int a;
};
```

It is equivalent to following class definition

```
class s
{
public:
   int a;
};
```

Class and Structure (Contd…)

- When class is defined as

```
class s
{
   int b;
};
```

It is equivalent to following structure definition

```
struct s
{
private:
   int b;
};
```

- Normally structures are used as C struct and classes as C++ classes.

Array of Objects

- Like normal variables we can create array of objects.
- For a following class

```
class test {
private:
   int data;
public:
   test(){} //default constructor
   test(int n){data=n;}//parameterized constructor
   //......};
```

- The array of object can be created as

```
test arr[10];
```

- Object array can be initialized like basic type array if one argument constructor is present, as

```
test t1[5]={3,4,7,10,12};
```

Array of Objects (Contd…)

- This initialization is equivalent to
```
test t1[5]={test(3),test(4),test(7),test(10),test(12)};
```
- When we have to invoke the two parameter constructor of the form
```
class test
{
private:
   int data1,data2;
public:
   test(int n1,int n2):data1(n1),data2(n2){}
   //......
}
```
- The array initialization list in this case must be specified as follows:
```
test t3[3]={test(2,5),test(7,9),test(4,8)};
```


Pointer to Objects

- Similar to the pointers to other standard variables, we can create pointer variable that will hold the address of the object.
- We can declare a pointer to object as
```
class_name  *pointer_to_object;
class_name  object_name;
```
- The address of an object to pointer variable can be assigned as
```
pointer_to_object=&object_name;
```
- When creating object dynamically, object pointer variable is used as
```
pointer_to_object=new class_name;//new class_name[size];
pointer_to_object=new class_name(args…);//constructor called
```
- Through the pointer to object we can access the object members as
```
pointer_to_object->member; //(*pointer_to_object).member;
```

The `this` Pointer

- Every C++ object has a implicitly defined pointer of its own type named `this` that points to itself
- The non static member functions of every object have access to the magic pointer named `this`.
- As static function can be accessed without creating object, they do not have access to `this` pointer.
- Let's see an example.

```
class test{
private:
    int a;
```


The `this` Pointer (Contd…)

```
public:
    func1(){…}
    func2(){
        cout<<this->a;  //or just a
        this->func1();  //or just func1()
    }
};
```

- This example shows the meaning of `this` pointer rather than its usage.

The this Pointer (Contd...)

- The following example shows the usage of this pointer.

```
class counter
{
private:
    unsigned count;
public:
    counter (unsigned n=0) {count=n;}
    counter inc(){
        count++;
        return *this; //or return counter(count);
    }
    int val(){return count;}
};
```

temporary nameless object

The statement `counter(count)` creates nameless object which is not efficient.

The this Pointer (Contd...)

- So the `counter::inc()` function can be used as

```
counter c1,c2(5);
c1=c2.inc();
```

The `counter::inc()` function increases the value of count member of `c2` and returns the new value of `c2`.

- This method is very useful with operator overloading as

```
c1=++c2;
```

In which the operator function has to return the value of the object.

- Another use can be when returning greater object among two as

```
t3=t1.greater(t2); //return greater object among t1 and t2
```

The `static` Data Members

- When a class is instantiated, memory is allocated for the created object, that is, memory is allocated for each member of the object.
- There are special type of data members declared as *static* for which the memory is not allocated during object creation.
- The static data members
  - belong to the class but not to object.
  - are declared in the class and are defined outside the class.
  - are stored separately.
  - are common to all the objects
  - are shared by all objects of that class
  - can be used before object creation or directly through the class.

The `static` Data Members (Contd…)

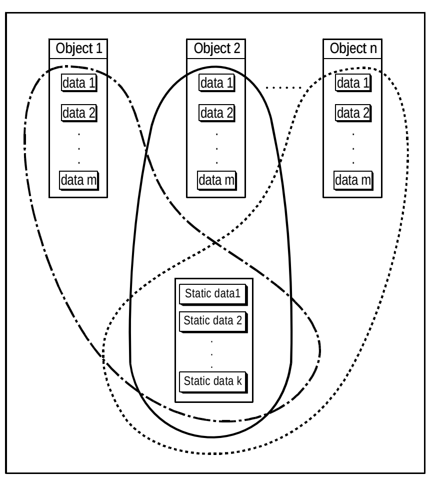


Figure: Memory model for static data

The `static` Data Members (Contd…)

- Let's see an example program

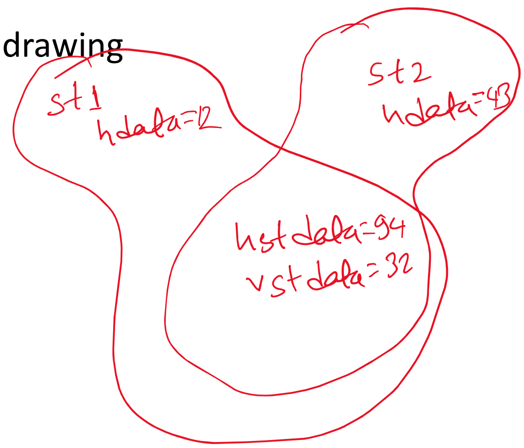
```
class stTest {
private:
    int hdata;
    static int hstdata;
public:
    static int vstdata;
    void setdata(int n1,int n2, int n3){
        hdata=n1;hstdata=n2;vstdata=n3;
    }
    void showdata(){
        cout<<"Private data= "<<hdata<<endl;
        cout<<"Private static data= "<<hstdata<<endl;
        cout<<"Public static data="<<vstdata<<endl;}
};
```


The `static` Data Members (Contd…)

```
int stTest::hstdata;  //initialized to zero when not initialized
int stTest::vstdata=5;
int main(){
    cout<<"Public st data from class="<<stTest::vstdata<<endl;
    stTest::vstdata=27; //Ok
    //stTest::hstdata=53; //Error, private static member
    stTest st1; //st1.vstdata can be done
    st1.showdata();
    st1.setdata(12,15,17);
    stTest st2;
    st2.setdata(43,94,32);
    st2.showdata();
    st1.showdata();
}
```

The `static` Data Members (Contd...)

- Show by drawing



The `static` Members Functions

- Similar to static data members, *static* functions belong to a class and not to any object.
- Static functions can be called without the object creation through the class using scope resolution operator as follows:
  `class_name::static_function();`
- The static member functions can also be called from objects using dot (`.`) operator.
- The static member functions can only access static data, so, are useful in accessing private static data.
- As static functions can be called before object creation or directly by the class they cannot access non static data member.

The `static` Members Functions (Contd…)

- Let's see the example of static member function

```
class element {
private:
    static int count;
    int data;
public:
    element (){count++; data=count; }
    ~element () {
        count--;
        cout<<"Destroying element with value"<<data<<endl; }
    static void showcount()      // static function
    {   cout<<"Number of elements are:"<<count<<endl; }
    void showdata()
    {   cout<<"The data is:"<<data<<endl; }
};
```


The `static` Members Functions (Contd…)

```
int element::count=0;
int main()
{
    element s1;
    element::showcount();
    element s2,s3;
    element::showcount();
    s1.showdata();
    s2.showdata();
    s3.showdata();
    return 0;
}
```

Constant Member Functions and Constant Objects

- We can make member functions in a class that does not alter values of its data members.
  - But still there can be some cases where we can accidently write code that changes the values of its data members.
- The C++ allows us to define member functions that guarantees not to change the data members value (eventually object's value) and prevents accidental alteration.
- These types of functions that guarantee not to change object's value (or data member value) are called *constant member functions*.
- Constant member functions are useful for constant objects.
  - Constant objects can only call their constant member functions.
- Constant member functions are declared by placing `const` keyword after the parameter list and before function body.

Constant Member Functions and Constant Objects (Contd…)

- Let's see the following code

```
class test
{
private:
    int data;
public:
    test(){data=0;}
    test(int n){data=n;}
    void setdata(int n) {data=n;}
    void showdata() const
    {   cout<<"data="<<data<<endl;}
};
```

Constant Member Functions and Constant Objects (Contd…)

```
int main()
{   const test t1(5);
    test t2(7);
    //t1.setdata(9);//Error, calling non constant function
    t1.showdata();//Ok, constant function
    t2.setdata(4);//Ok
    t2.showdata();//Ok
    return 0;
}
```

- A constant object cannot call a non constant member function even if it is does not alter the values
- A non constant object can call constant as well as non constant member functions

The `const_cast` Operator

- In some cases, classes can use data which are used internally for the manipulation of the objects but are not visible to the user by any means.
- Changing the value of these members by the constant function may not change the meaning of the constantness to the user.
- In rare cases, constant functions need to change the value of internally used unobservable data members.
- ISO C++ provides keyword `const_cast` operator which is used in the constant function to remove the constantness and update those types of members.

The `const_cast` Operator (Contd…)

- Let’s see the example

```
class test {
private:
    int data1;
    int data2;//changeable attribute
public:
    test(){data1=0;data2=0;}
    test(int n){data1=n;data2=0;}
    void showdata() const
    {   test *tp=const_cast<test*>(this);
        tp->data2=tp->data2+1;
        cout<<"data="<<data1<<endl;
        cout<<"func called for "<<data2<<" times"<<endl;}
};
```


The `const_cast` Operator (Contd…)

```
int main()
{
    const test t1(5);
    test t2(7);
    t1.showdata();//undefined
    t2.showdata();//Ok
}
```

- The behaviour of `const_cast` operator is undefined if the object itself is declared as `const`.

The `mutable` Class Members

- Using the `const_cast` operator it is possible to remove constantness of object or member.
- The `const_cast`'s behavior is implementation dependent when the object is constant.
- As an alternative to `const_cast` operator the data members themselves can be declared as changeable even with constant function and objects.
- ISO C++ provides keyword `mutable` to specify changeable member.
- A mutable data member is always modifiable even with constant member function or constant object.

The `mutable` Class Members

- So the above code can be written as

```
class test {
private:
    int data1;
    mutable int data2;//changeable attribute
public:
    //…
    void showdata() const
    {
        data2=data2+1;
        cout<<"data="<<data1<<endl;
        cout<<"func called for "<<data2<<" times"<<endl;}
};
```

Friend Functions

- Private members of an object are not accessible from the code outside the class.
- The private members can be accessed indirectly through the public member functions.
- Sometimes this feature leads to inconvenience.
  - For example, if we want to use a function to operate on objects of two different classes, then a function outside a class should be allowed to access and manipulate the private members of the class.
- The C++ allows us to access the private members of an object through the concept of *friend functions*.

Friend Functions (Contd…)

- To declare that some function is a friend of a class the function is declared in the class by prefixing with a keyword `friend`.
- A friend declaration can be placed in either private or public part of the class declaration.
- The function mentioned as a friend of a class can be a global function or function in any scope.

```
class test
{
   //......
   friend return_type friendfunc(prameters…);//declaration
};
return_type friendfunc(prameters…){…} //definition
```

Friend Functions (Contd…)

- The friend function can be used as follows

```
class test
{
private:
    int data;
public:
    test(int v=0){data=v;}
    int val(){return data;}
    friend void inc(test &a);
};
```


Friend Functions (Contd…)

```
void inc(test &a){
    a.data++;
}
int main()
{
    test t(4);
    cout<<"Value before increment: "<<t.val()<<endl;
    inc(t);
    cout<<"Value after increment: "<<t.val()<<endl;
    return 0;
}
```

- Friend functions are also useful when we have to bridge between two classes.

Friend Classes

- Similar to friend function, any class can be friend of another class
- When any class is declared as friend of another class then all the member function of that class can access the private members of another class.
- A class can be declared as friend as

```
class B;
class A
{
     //......
     friend class B; //B is declared as friend of A
};
```


Friend Classes (Contd...)

- Let's see an example

```
class second;
class first
{
private:
    int data;
public:
    first(int v=0){data=v;}
    int val(){return data;}
    friend class second;
};
```

Friend Classes (Contd…)

```
class second
{   //...
public:
    void inc(first &a) {a.data++;}
};
int main()
{
    first f(5);
    second s;
    cout<<"Value before increment: "<<f.val()<<endl;
    s.inc(f);
    cout<<"Value after increment: "<<f.val()<<endl;
    return 0;
}
```


Friend Classes (Contd…)

- If class B is declared as friend of class A then class B can access private members of class B but reverse is not true.
- If both of the class have to access the private members of one another then both classes have to be declared as friend of one another.
- Instead of making whole class as friend, specific member function of a class can be made friend.