```
class myclass
{
    int m1;
public:
    int m2;
    myclass(int a, int b): m1(a),
    m2(b) { }
    int getmember() { return m1; }
    int setmember(int i)
    {
        m1 = i;
    }
    void print()
    {
        cout << "private :" << m1 <<
        "public" << m2 << endl;
    }
};
int main()
{
    const myclass consObj(5, 6);
    cout << consObj.getmember() << endl;
    consObj.m2 = 8;
    consObj.setmember(7);
    consObj.print();
}
```


our are error

Since const Object can not invoke non const method

(12) Constant member function

- To declare a constant member fn we use keyword const betn the fn header and the body like.

```
void print () const { cout<<my<<endl;}
or
```

- When we declare a constant member fn then the this pointer which is passed gets changed in its signature like

const myclass * const this;

on hence can be invoked by constant object

- In constant member fn no data member can be changed.

```
void setmember( int i) const
{ m2 = i;} //Not allowed.
gives an error
```

→ non constant. object can invoke const member fn

→ constant object can only invoke const member fn.

→ All member fn that do not need to change an object must be declared as constant member fn

(14) Changing getmembers and print in previous

example

```
int getmember() const
{ return m1;
}
void print() const
{ cout<<"private:"<<m1<<"public:
  "<<m2;
}
};
```

Now

```
int main()
{
    myclass obj1(0,1) // Non-constant Object
    const myclass obj2(5,6) // constant Object.

    cout<< obj1.getmember()<<endl;
    obj1.setmember(2);
    obj1.m2=3
    obj1.print();

    obj2.print()
    cout<< obj2.getmember()<<endl;
    obj2.print()
```

O/P: 0
2,3

O/P: 5
5, 6

(29) Constant Data members

empID, DOB

A constant data member cannot be changed even in a non-constant Obj

A constant data member must be initialized or the initialization list

```
class myclass
{   const int cpm;
    int pm;
public:
    const int &cpubm;
    int pubm;
    myclass(int a, int b, int c, int &d);
    cpm(a), pm(b), cpubm(c),
    &pubm(d) { }
    get int getprivm()
    { return pm;
    }
    get int getcprivmem()
    { return cpm;
    }
    void setprivate(int i)
    { pm = i;
    }
    void setcpriv(int i)
    { cpm = i; // error
    }                cpm is
};                   constant
```

(139)

```
myclass myObj (1, 2, 3, 4);
cout << myobj.getprivate() << endl;
cout << myobj.getCprivate() << endl;
cout << myobj.Cpubm << endl;
myObj.Cpubm = 3 // error
cout << myobj.pubm << endl;
}
```

mutable data members

While a constant data member is not changeable even in a non-constant object, mutable data member is changeable even in a constant object.

- mutable is provided to model logical (semantic) constness against the default Bitwise (syntatic) constness in C++.

Note

- mutable is applicable only to data members and not to a variable
- Reference data members cannot be declared as mutable
- Static Data member cannot be declared as mutable.
- const Data member cannot be declared as mutable.

(13) If a Data member is declared mutable, then it is also possible to assign a value to it from a const member fn.

ex:

```
class myclass
{   int mem;
    mutable int mutmem;
public:
    myclass(int a, int b): mem(a),
    mutmem(b) { }
    int getmem() const
    {   return mem;
    }
    void setmem(int i)
        mem = i;
    int getmutmem() const
    {   return mutmem;
    }
    void setmutmem(int a) const
    {   mutmem = a;
    }
};
```

It can change mutmem

int main()
{
    const myclass constObj(1, 2);
    cout << constObj.getmem() << endl;
    constObj.setmem(3) ✗ error
    cout << constObj.getmutmem() << endl;
    constObj.setmutmem(4);
}

since mutmem is mutable

Logical vs bitwise constantness

* const in C++, makes bitwise constant. One an Object is constant no part of it (actually no bits of it can be changed after construction or initialization.

* However, while programming we often need an object to be logically constant. That is the concept represented by the Object should be constant But it's representation need more Data members for computation and modelling these have no reason to be constant

* mutable allows such surrogate Data members to be changeable in a (bitwise) constant Object. to model logically const Object.

To use mutable we should look for

* A logically constant concept
* A need of Data members outside the representation of the concept, But are needed for computation.

When to use mutable Data members

* Typically when a class represents a constant concept and it computes a value for first time and caches the result

for future use.

ex: slide

when not to use variable.

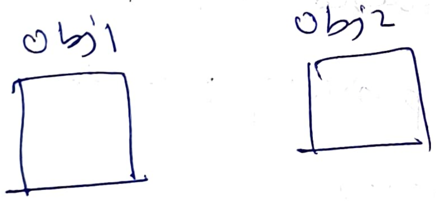
Static Data member

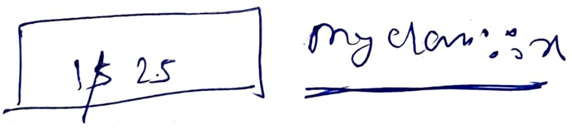
- is associated with class not with object (No matter how many instances of class is created, there will be only one instance of static data member of class)
- is shared by all objects of class.
- static data member will exist even when no objects are created for the class
- needs to defined outside the class scope (in addition to declaration within class) to avoid linker error
- must be initialized in a source file.
- is constructed before main starts and destructed after main ends
- can be private / public
- can be accessed
  - with class name followed by (::)
  - as a member of any object of class
- virtually eliminated any need for global variable in OOPs env.
- must be initialized in global scope and static data members are initialized during program startup.

ex:
```
class myclass
{  static int x;   // Declare static data member
public:
    void get() { x = 15; }
    void print()
    {  x = x+10;
       cout<<"x="<<x<<endl;
    }
};
int myclass::x = 0   // Define static data member
int main()
{  myclass obj1, obj2;
   obj1.get(); obj2.get();
   obj1.print(); obj2.print();
   return;
}
```

O/P: x = 25
x = 35

O/P





ex. print fact

(135) static member function — if static data member becomes private how do I manipulate it

- does not have this pointer. (not associated with any objects).
- can not ~~access~~ invoke non static data members
- can be accessed
  - with class Name :: fn Name.
  - as a memb of any obj of class
- is needed to read / write static data members
  - (a) for encapsulation static data member should be private
  - (b) get-set idioms is built to access (static mem fn is public)
- may initialize static data member before any class obj creation.
- cannot co-exist with a non-static version of same fn i.e non static and static memb of fn with same name not possible
- can not be declared as const.

ex. slide

Singleton class — kind of design pattern

- A class is called a singleton if it can have only one instance at a particular time

ex: president of India.
pm

How to implement a singleton class:
make the constructor private.

How to restrict the user can't create only one instance.
By using static private member and static member fn

exp: Slides

friend function

- has access to the private and protected members of the class (breaks the encapsulation)
- must have its prototype included within the scope of the class preceded with keyword friend
  - does not have name qualified with class scope
- is not called with an invoking object of the class
- a fn can be declared friend of more than one class

What all fn can be friend fn:

(i) A global fn
(ii) Any member fn of dist class can be friend of another class

(iii) A function template.

When we require to manipulate the internals of class we require friend fn

ex:

```
class myclass
{
    int data;
public:
    myclass(int i); data(i) {}
};
void Display(const myclass &a)
{
    cout<< data << a.data <<endl;
}
```

error since data is private member

```
int main()
{
    myclass Obj(10);
    Display(Obj);
    return 0;
}
```

ex:2. Declare Display fn as friend of myclass.

```
class myclass
{
    int data;
public:
    myclass(int i): data(i){}
    friend void Display(const myclass &a);
};
void Display(const myClass &a)
{
    cout<<"data" <<"a.data<<endl;
```

```
int main()
{
    MyClass Obj(10);
    Display(Obj);
    return 0;
}
```

ex-1, 2 (slides)

ex.2 How a friend one class can be made friend of another class.

```
class Node;
class List
{   Node *head;
    Node *tail;
public:
    List(Node *h=0) : head(h),
    & tail(h) { }
    void display();
    void append(Node *p);
};
class Node
{   int info;
    Node *next;
public:
    Node(int i) : info(i), next(0){}
    friend void List::display();
    friend void List::append();
};
```

delete search

void List :: display()
{
}

void List :: append(Node *p)
{
}

Note: using friend fn we can actually link up two such classes which are somewhat related but completely independent.

friend class

A friend class of a class

- has access to the private and protected member of class (breaks the encapsulation)
- Does not have its name qualified with the scope (not a nested class)
- can be declared friend in more than one class

friend class can be

- class
- class template

all member fn of a class which is friend of another class have access to private and protected data member of another class.

ex: since all fn of List is friend of Node class making List class as friend of Node class.

like:

```
class Node
{
    int info;
    Node *next;
public:
    Node(int i) : info(i), next(0) {}
    friend class List;
};
```

Important points

* Friendship is neither commutative nor transitive
* Friend tend to break data hiding and should be use judiciously like
  1. A fn needs to access the internals of two class
  2. A class is built on top of another
  3. certain situation of operator overloading (like streaming operator

(14)

Operator overloading for UDT

* Operator overloading helps us to build complex algebra for UDTs much in the same line as is available for built in types.

Non member operator function (operator overloading for non member)

↓

A non member operator fn can be
(a) global fn
(b) friend "

ex: **Binary operator**

```
MyType a,b;

~~MyType operator+( MyType a, MyType b)~~

MyType Operator +( const MyType &a,
    const MyType &b) // global.

friend MyType operator +( const
    MyType &a, const MyType &b)
```

Unary Operator

```
MyType Operator ++ (const MyType &);
friend MyType operator ++( const MyType &)
```

Note: parameters may not be constant and may be passed by value. The return may also be reference and may be constant.

(a+b) same as operator+(a,b)

a=b operator=(a,b)

++a operator++(a)

a++ operator++(a,0, int)

C=a+b operator=(operator+(a,b))

Note: (i) For a member operator fn invoking object is passed implicitly as left operand but the right operand is passed explicitly

(ii) For non member (or operator fn (global/friend) both the operands are explicit

ex. Operator Overloading circle.

eg.1

```
struct complex
{ double re;
  double im;
};
Complex operator+( Complex &a,
                   Complex &b)
{
    complex t;
    t.re = a.re + b.re;
    t.im = a.im + b.im;
    return t;
};
```

(43) Complex d1, d2, d

d1.re = 2.14 , d2.re = 3.14

d1.im = 3.12 , d2.im = 5.12

d = d1 + d2

↑

It will call Operator+(d1, d2)

ex.2 make the data member private and provide get-set and Overload + operator as global fn. But still we are expecting internals of class. So Overload operator as member fn of class

Ex:

```
class complex
{   double re, im;
public:
   Complex(double a=0.0, double b=0.0): re(a), im(b) {}
   ~Complex() {}
   void display()
```

It will take this ptr implicitly

```
Complex operator +(const complex &c)
{
   complex t;
   t.re = re + c.re;
   t.im = im + c.im
   return t;
};
};
```

Complex c1(4.1, 3.25), c2(4.12, 3.9), c3

c2 = c1 + c2

Overloading operator = (copy assignment operator)

* If we are not overloading operator = then the compiler will provide free copy assignment operator (i.e overloading of operator =)

* Overloaded operator = may choose both deep and shallow copy for pointer members

* compiler provided free operator = makes only shallow copy

* If the constructor uses operator new, operator = should be Overloaded

* If there is a need to deep define a copy constructor then operator = must be overloaded and vice versa

Ex: Overloading array Operator

```
class myclass
{
    int data;
public:
    myclass(int d) : data(d){ }
```

~~Myclass ++~~

(105)

myclass operator ++( )

{ ++data; return *this; }

myclass operator ++(int)

{ myclass t(data); ++data; return t; }

myclass obj(10);

myclass obj2 = obj1++;

obj2 = ++obj1;

we could write any function ordinarily instead of ++ operator

assigns the same value and then increments returning the new obj with same value and increment is original object.

extending operator +

complex d1(2.5, 3.2) d2(1.6, 3.3) d3

d3 = d1 + d2

d3 = d1 + 5.2 // d3 = where 5.2 treat 5.2 im = 0

d3 = 5.2 + d2 // im = 0

Issue-2 (148) Overloading operator << and operator >>

Complex d

cin >> d → istream object

cout << d → ostream object

extending operator + using global fn

Ex: class Complex

{ public: double re, im;

explicit Complex(double r=0, double i=0) : re(r), im(i) { }

void disp() { } };

overload Complex operator+(const Complex &a, const Complex &b)

{ return Complex(a.re+b.re, a.im+b.im); }

Complex operator+(const Complex &a, double d)

{ Complex b(d); return a+b; }

create temp object and use overload 1

Complex operator+(double d, const Complex &a)

{ Complex b(d); return (a+b); }

create temp obj and use overload 1

(142) Complex d1(2.5, 3.2), d2(1.6, 3.3), d3

d3 = d1+d2

d3 = d1 + 4.2 — Issue it

d3 = 5.5 + d2 — because encapsulation

expanding operator + with member fn

```
class complex
{
    double re, im;
  public:
    explicit complex (double r = 0, double i = 0): re(r), im(i) {}
    complex operator + (const complex &c)
    {   return complex( re + c.re, im + c.im)
    }
    complex operator + (double d)
    {   complex b(d);
        return *this + b
```

complex d1(2.5, 3.2) d2(1.6 3.3), d3

d3 = d1+d2

d3 = d1 + 6.2

d3 = 4.2 + d2 → Not possible

needs on operator of left.

extending operator + with friend function

```
class complex
{   double re, im;
public:
    explicit complex( double r=0, double i=0): re(r), im(i) {}
    friend complex operator+ (const complex &a)
    {
        return complex (re+a.re, im+a.im);
    }
    friend complex operator+ (const complex &a, double d)
    {
        complex b(d);
        return a+b;
    }
    friend complex operator+ (double d, const complex &a)
    {
        complex b(d);
        return a+b;
    }
};
complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
d3 = d1+d2
d3 = d1+4.2
d3 = 3.5+d2
```

we do not want to use implicit casting. It tells the compiler that do not do any casting for this object

(149) Overloading I/O operators operator<<, operator>>

complex d1, d2

cout<<d1<<d2 //((cout<<d1)<<d2

↑ chain of output

Eg: Signature of operator to overload this

global fn

ostream & operator<< (ostream & os, const complex &a);

member fn in ostream

ostream & ostream :: operator<< (const Complex &a)

using fn in complex

ostream & Complex :: operator<< (ostream & os)

// return by reference for ostream object is used so that chaining would work

ex:

class complex

{ public: double re, im;

complex(double r=0, double i=0) : re(r), im(i) { }

};

ostream & ostream<< (ostream &os, const complex &a)

{ os<< a.re << "+j" << a.im << endl;

return os;

}

```
istream & operator>>(istream &is,
                     complex &a)
{
    ~~is<<a.re<<~~
    is>>a.re>>a.im
    return is;
}
int main()
{
    complex d;
    cin>>d;
    cout<<d;
}
```

Overloading << operator with member fn is not

* Case-I operator << is a member in ostream class.

~~Ostream & Operator<<~~

Ostream & Ostream :: operator << (const Complex &a)

Ostream is C++ standard lib. and we are not allow to edit this lib to include this

Case-II operator << is a member fn in Complex class

Ostream & Complex :: operator << (Ostream &os);

to invoke this → d << cout

Semantic is wrong.