

(126)

class myclan

{ int m1;

public:

int m2;

myclan(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 myclan* consObj(5, 6)

cout << consObj->getMember() << endl;

consObj->m2 = 8 }

consObj->setMember(7)

our code error } consObj->print();

~~Since const Object cannot invoke non
const method~~

const & const member function

To declare a constant member fn we use keyword const b/w the fn header and the body like -

void print() const {
cout<<myvar
endl;}

When we declare a writing member fn then return's pointer which is passed gets changed in the signature like

const myClass * const this;
or hence can be invoked by
constant object

In constant member fn no data
member can be changed.

void setmember(int i) const
i = m2 = 1; // 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

⑩ changing get members and print in private 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();

2 3

Obj2.print();

cout << Obj2.getmember() << endl;

Obj2.print();

Obj2.print();

5 6

Q) constant data members

Ex: emp ID, DOB

A constant data member can not be changed even in a non-constant obj

A constant data member must be initialized at the initialization line

class myclass

h. 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),

Cpubm(d) } }

get int getpm()

{ return pm;

}

get int getcpubm()

{ return Cpm;

}

void setpm(int i)

{ pm = i;

}

void setcpubm (int i)

{ Cpm = i; // error

}

Cpm is ~~not~~

}

Constant

(3) myclm myObj(1, 2, 3, 4);
cout << myObj.getprimval() << endl;
cout << myObj.getprimval() << endl;
cout << myObj.getprimval() << endl;
myObj.getprimval = 3 // error
cout << myObj.getprimval() << 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 logically (semantic) constraints against the default Bitwise (syntactic) constraints in C++.

More

mutable is applicable only to Data members and not to a variable.

→ Reference Data members can not be declared as mutable.

→ static Data member can not be declared as mutable.

→ const Data member can not be declared as mutable.

Q) If a data member 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;
    }
}
```

The change in mutMem
since mutMem is mutable

```
int main()
{
    const myclass constObj(1,2);
    cout << constObj.getMem() << endl;
    constObj.setMem(3) X error
    cout << constObj.getMutMem() << endl;
    constObj.setMutMem(4) ✓
}
```

Logical vs bitwise constants

(132)

* While in C++, models bitwise constant one on object's constant no part of it (actually no bit) 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 its 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 work 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.

(133)

~~ex: slide~~

when not to use mutable.

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 static or static data member of class)

is stored by all objects of class.

static data members 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 start and destroyed after main ends~~

~~can be private / public~~

~~can be accessed~~

with class name followed by (>:

as a member of any object of class

virtually eliminates any need for global variable in OOPs thus

~~must be initialized in global scope and static data members are initialized during program startup.~~

ex: class Myclass

(134)

L

static int x; // Define static Data member

public:

void get() {
 n = 15; }

void print()

{ n = 10;

cout << "n = " << endl;

}

};

int myclass::x = 0; // Define static Data member

int main()

{ Obj1 obj1, Obj2;

Obj1.get(); Obj2.get();

Obj1.print(); Obj2.print();

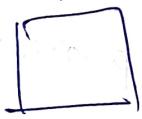
return;

}

Obj



Obj2



Obj; 25

$n = 25$

$n = 35$

$n = 35$

if 25 My class::n

ex: print func

int print()

{ cout << n;

 return 0;

(3) static member function ~~at static data member~~
becomes private how do

to manipulate it

- does not have this pointer. (not associated with any object).
- can not access non-static data members
- can be accessed
 - with class Name:: fnName.
 - as a member of any Obj creation

— is needed to read / write static data members

(a) ~~for~~ for encapsulation static data member should be private

(b) get-set idiom is built to access static members from 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 members for non with same name not possible

— can not be declared as const

ex. sides

Singleton class - kind of Design pattern (136)
A class is called a Singleton if it contains
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 others to user can't create
only one instance.

By using static Data member and static member

fn

Ex: 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 prefixed
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 can & can't be friend fn:

(i) A global fn

(ii) Any member fn of diff class
can be friend of another class

(iii) A function template.

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

Ex:-

```
class Myclass
{
    int data;
public:
    Myclass(int i); ~Myclass();
};
```

void display(const Myclass &a)

h. cout<<data <<(a.data << endl);
} error since data
is private member

int main()

h. Myclass obj(10);
 display(obj);
 return 0;

y

Ex. 2. ~~make~~ Define display fn as friend of myclass

class Myclass

h. int data;

public:

```
Myclass(int i); ~Myclass();
```

Ex:-

friend void display(const Myclass &a);

};

~~Display~~

void display(const Myclass &a)

h. cout<<"data" <<(a.data << endl);

int mouse,

(138)

```
    myname obj(10);  
    display(obj);  
    return 0;  
}
```

Ex-1.2 (sliders)

Ex-2 now a lot of 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> info(), next();

friend void List::display();

void void

friend void List::append();

```
void List::display()  
h  
? List::append(Node*)  
void
```

(139)

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

friend class

~~if a friend class of a class~~

~~has access to the private and
protected members of class (breaks
the data 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~~

~~other members of
any member fn
of other class which~~

~~is friend of or me~~

~~Class have access to
private and protected Data members of
another class.~~

Ex since our ~~Node~~. ~~node~~ fn of List is friend of
Node class making List class as friend of
Node class.

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 show & be use judiciously like~~~~(1) A fn needs to own the members of two class~~~~(2) A class is built on top of another~~~~(3) certain situations propose for~~~~overloading (like streaming operator)~~

4

Operator overloading for user defined types

(4) ~~* Operator overloading helps us to build complex class for user defined type in the same lines as available for built-in types.~~

Non member operator & function (operator overloading for non member)

An non member operator fn can be

- (a) global fn
- (b) friend "

Binary operator

MyType a, b;

~~MyType operator+(MyType a, const 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

(142) $a+b$ goes on operator $+(a, b)$

$a = b$ operator $= (a, b)$

$+ * a$ operator $* (a)$

$a++$ operator $++(a)$, i.e.

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

- Note:
- ① For a member operator $\&$ involving object is passed implicitly as left operand but the right operand is passed explicitly
 - ② For non-member $\&$ operator $\&$ (global friend) both the operands are explicit

ex. Operator Overloading example.

eg. 1 struct complex

 double re;

 double im;

{

 Complex operator + (Complex & b)

 Complex & b)

h

 complex &)

 for a.ref.b.ref;

 f.im = a.im + b.im;

 return f;

{

(43) Complex d1, d2, d

$$d1.r = 2.14, d2.r = 3.14$$

$$d1.i.m = 3.12, d2.i.m = 5.12$$

$$d = d1 + d2$$

↑
It will call Operator

d1, d2

Ex: make the base member private
and provide get-set and Overload +
operator as global fn. But still we are
expiring members of class. So Overload
operator on member fn of class

Ex:

class complex :

{ double re, im;

public :

Complex(double a=0.0, double
b=0.0); re(a), im(b) }

\sim Complex()

void display()

Complex operator + (Complex
const Complex & c)

{

Complex t;

$$t.re = re + c.re;$$

$$t.im = im + c.im$$

return t;

3

4

return t;

(144)

Comparison (i_1)², i_2 (25), (i_2 (i_1)², 5), (i_2

$i_2 = i_1 + i_2$

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 between deep and shallow copy for pointer members

* Compiler provided free operator = makes only shallow copy

~~* If the consumer 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 copy Operator

class myclass

{ int data;

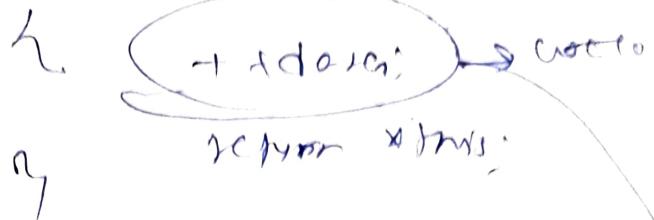
public:

myclass(int d) : data(d) {}

~~Reclass ++~~

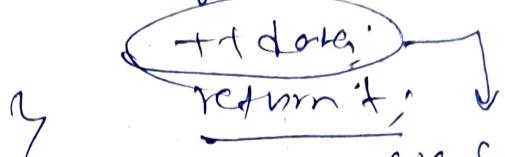
~~my class~~ non

my class & operator ++()



my class operator ++(int)

my class + (data)



my class Obj (18)

my class Obj2 = Obj1 + f.

Obj2 = ++Obj1;

Assigning the same value and then incrementing so returning the new obj with some value and increment is original Object.

extending operator +

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

$$d3 = d1 + d2$$

$$d3 = d1 + s.2 // d3 = \text{where } \boxed{s = 2}$$

mean s2 m=0

$$d3 = \underline{\underline{4.2 + d2}} // \underline{\underline{m=0}}$$

Lesson-2 (17) Overload operators & operator overloading

complex
 ↗
 (cout) ↗
 ↗ istream object

→ ostream object
 extending operator + coin global fn

Ex. class complex

```

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

        void disp() const { }
    }
  
```

?;
 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 obj
 and use
 and overload
 create temp obj
 and use
 and overload
 complex operator + (const complex& a,
 double d, const complex& b)
 complex b(d);
 return a+b;

(Q2) complex d1(2.5, 3.2), d2(1.6, 3.3), d3

$$d3 = \underline{d1 + d2}$$

$$d3 = d1 + \underline{d2}$$

$$d3 = 2.5 + d2 \quad \begin{cases} \text{Now it} \\ \text{breaks encapsulation} \end{cases}$$

expanding operator + with operator 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 re*d+b

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

$$d3 = d1 + d2$$

$$d3 = d1 + \underline{d2}$$

$$d3 = 2.5 + d2 \rightarrow \text{Not possible}$$

need an operator of left.

Extending Operator + with friend function (148)

class complex

↳ double re, im
 we do not want to have
 public implicit conversion. It will be
 compiler's job to do not
 do any conversion.
 explicit complex(double r=0, double i=0);

i = 0; re(r), im(i) {}

friend complex operator+(const complex &a)

↳ return (re+a.re)

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

$$d3 = d1 + d2$$

$$d3 = d1 + 4.2$$

$$d3 = 3.5 + d2$$

(14) Overloading LO Operator operator <<,

operator >>

complex d1, d2

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

↑ chain of output

global En: signature of operator to overload this

operator <<

ostream & operator << (ostream & os, const com
plex & a);

members in ostream

ostream & operator << (ostream & os, 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 << " + " << a.im << endl

return os;

istream & operator>>(istream & is, 150
complex &a)

↑

~~is & a~~

i>>a; a>>a::im

return is;

↓

int main()

↑

complex d;

cin>>d;

↓

cout<<d;

Overloading << operator with member fn in class

* 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 allowed to edit this lib to include this

case-II Operator << is a member fn in Complex class

class

ostream & Complex &:: operator <<
ostream & o;

↓

to invoke this

delecon

semantic is wrong.