



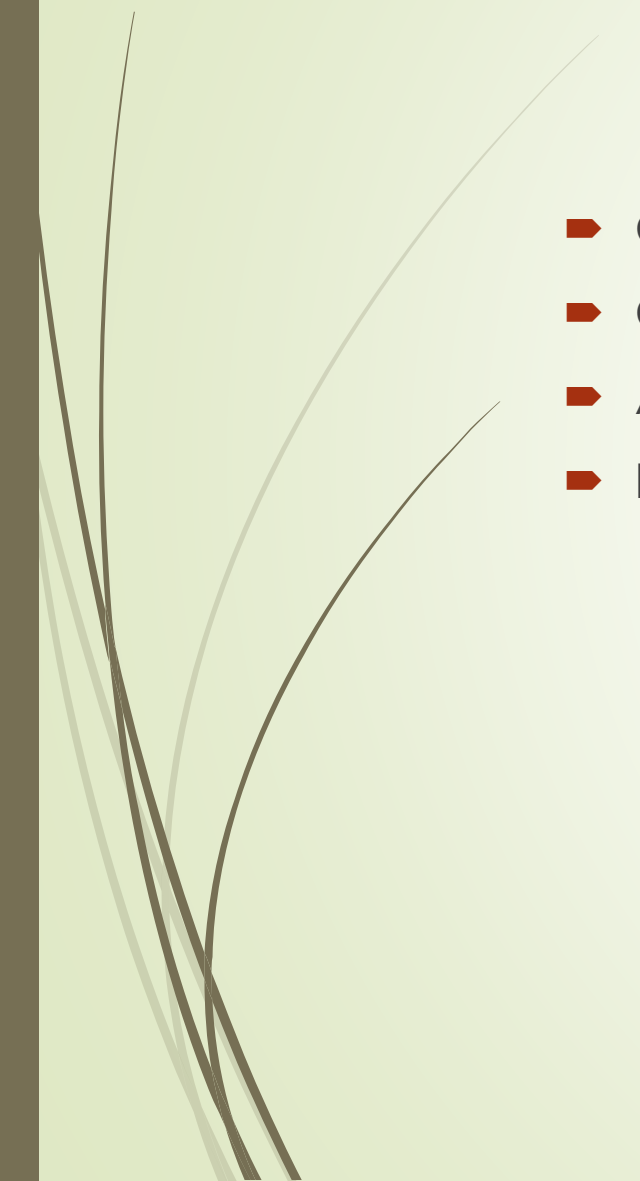
STAT 598W Object Oriented Programming in C++

Cheng Li

Purdue University



Class and Objects

- Class is similar as data
 - Class may contain data members and functions.
 - An object is an instantiation of a class.
 - In terms of variables, a class is the type, and an object is the variable.
- 



About Reference

- An alias of another variable.
- Always need initialization when defined.
- Example: used as parameters for the function to swap two numbers.
- Can be used as return value.

```

1  #include<iostream>
2  using namespace std;
3
4  class Rectangle {
5      int width, height;
6  public:
7      void SetValue(int w, int h) {
8          width = w;
9          height = h;
10     }
11     int CalculateArea() {
12         return width * height;
13     }
14     int & GetWidth() {
15         return width;
16     }
17     int & GetHeight() {
18         return height;
19     }
20 };

22 int main() {
23     Rectangle a;
24     //Define an object, named a, which is of Rectangle type.
25     a.SetValue(1, 2);
26     cout << a.CalculateArea() << endl;
27     //Get access to member functions.
28
29     Rectangle *p = &a;
30     //Define a pointer to a.
31     p->SetValue(2, 3);
32     cout << p->CalculateArea() << ' ' << a.CalculateArea() << endl;
33
34     Rectangle &b = a;
35     //Define a reference of a.
36     b.SetValue(3, 4);
37     cout << b.CalculateArea() << ' ' << a.CalculateArea() << endl;
38     //Get access to member functions through reference.
39
40     a.GetHeight()++;
41     a.GetWidth()++;
42     cout << a.CalculateArea() << endl;
43 }

```

```

2
6 6
12 12
20

```

Write Member Functions Outside of Class Definition

➡ '::' is called the scope operator.

```
22 class Rectangle2 {  
23     int width, height;  
24     public:  
25         void SetValue(int, int);  
26         int CalculateArea();  
27 };  
28 void Rectangle2::SetValue(int w, int h) {  
29     width = w;  
30     height = h;  
31 }  
32 int Rectangle2::CalculateArea() {  
33     return width * height;  
34 }
```



Access Specifier

- The access specifier includes: private, public or protected.
 - private: accessible only from within other members of the same class (**It can be accessed by another object of the same class**).
 - protected: accessible from other members of the same class and from members of their derived classes.
 - public: accessible from anywhere where the object is visible.
- No restriction on the order and times each specifier appears.
- Without specified, members are private.
- **Data abstraction:** providing only essential information to the outside world and hiding their background details.



Function Overloading

- Same function name, different input parameter list.
- It is not allowed that two functions only differ from return type.
- It is not allowed that two functions only differ from if a parameter is of const type or not.
- Need to avoid ambiguity with functions with default values.
- Member functions, constructors and ordinary functions can all have overloading.


```

5 //Constructor, Destructor
6 class complex {
7     double re;
8     double im;
9 public:
10    complex() {
11        re = im = 0;
12        cout << "Constructor 1 is called!" << endl;
13    }
14    complex(double _re) {
15        re = _re;
16        im = 0;
17        cout << "Constructor 2 is called!" << endl;
18    }
19    complex(double _re, double _im) {
20        re = _re;
21        im = _im;
22        cout << "Constructor 3 is called!" << endl;
23    }

```

```

24    complex(complex& a) {
25        re = a.re;
26        im = a.im;
27        cout << "Copy constructor is called!" << endl;
28    }
29    ~complex() {
30        cout << "Destructor is called!" << endl;
31    }
32 };
33
34
35 int main() {
36     complex a;
37     complex b(1);
38     complex c(1, 2);
39     complex d(c);
40     complex e = d;
41     complex *p = new complex(1);
42     delete p;
43     return 0;
44 }

```

```

Constructor 1 is called!
Constructor 2 is called!
Constructor 3 is called!
Copy constructor is called!
Copy constructor is called!
Constructor 2 is called!
Destructor is called!
Destructor is called!
Destructor is called!
Destructor is called!
Destructor is called!

```




Constructor



- A special type of member function, same name as the class, no return value (**not even void type**).
- Constructor is called when an object is created. After the object is created, constructor cannot be called again.
- The purpose is to initiate members.
- If no constructor is defined, compiler will generate a default constructor, who has no parameter and does nothing.
- If a constructor is defined, no default constructor will be generated.
- A class can have several constructors.

Constructor and Initialization of Array

- `ClassName a[10];`
 - `//Call constructor without parameter.`
- `ClassName a[10] = {1, 2};`
 - `//Call constructor with one parameter. The rest initialized by non-parameter constructor.`
- `ClassName a[10] = {ClassName (1, 2), ClassName (3, 4)};`
 - `//Call constructor with two parameters. The rest initialized by non-parameter constructor.`
- `ClassName a[10] = {ClassName (1, 2), 3, 4};`
 - `//Misture.`



Constructor and Initialization of Array

- ▶ `ClassName* p = new ClassName[10];`
 - ▶ `//Call constructor without parameter.`
- ▶ `ClassName* p[10] = {new ClassName(1), new ClassName(2, 3)};`
 - ▶ `//Call constructor without parameter.`



Copy Constructor

- Syntax:
 - `ClassName(className& _input);`
 - `ClassName(const ClassName& _input);`
- The following is not allowed as a copy constructor:
 - `ClassName(ClassName _input);`



Copy Constructor



- If no copy constructor is defined, then a default copy constructor is defined by compiler, which does nothing.
- If a copy constructor is defined, no default copy constructor is generated.
- If a copy constructor is defined, no default constructor is generated.
- If a constructor is defined but no copy constructor is defined, then no default constructor is generated but a default copy constructor is defined.



Copy Constructor

- Three situations that copy constructors are called.
- (1) `ClassName Object1 (Object2);`
- (1') `ClassName Object1 = Object2;`
- **After “Object1” is defined, “Object1 = Object2” does not call the copy constructor. It calls the overloading of “=”.**
- (2) Pass an object by value, as parameter, to a function.
- (3) Use an object as return value of a function.



Constant Reference

- To avoid calling copy constructor, we can use reference.
- If you want to make sure that you will not modify the original value, can use const reference.
- Example:
 - `void f(const ClassName & a);`



Conversion Constructor

- Constructor with one parameter, but not copy constructor, is usually a conversion constructor.
- Example:
 - `Complex(double _re);`



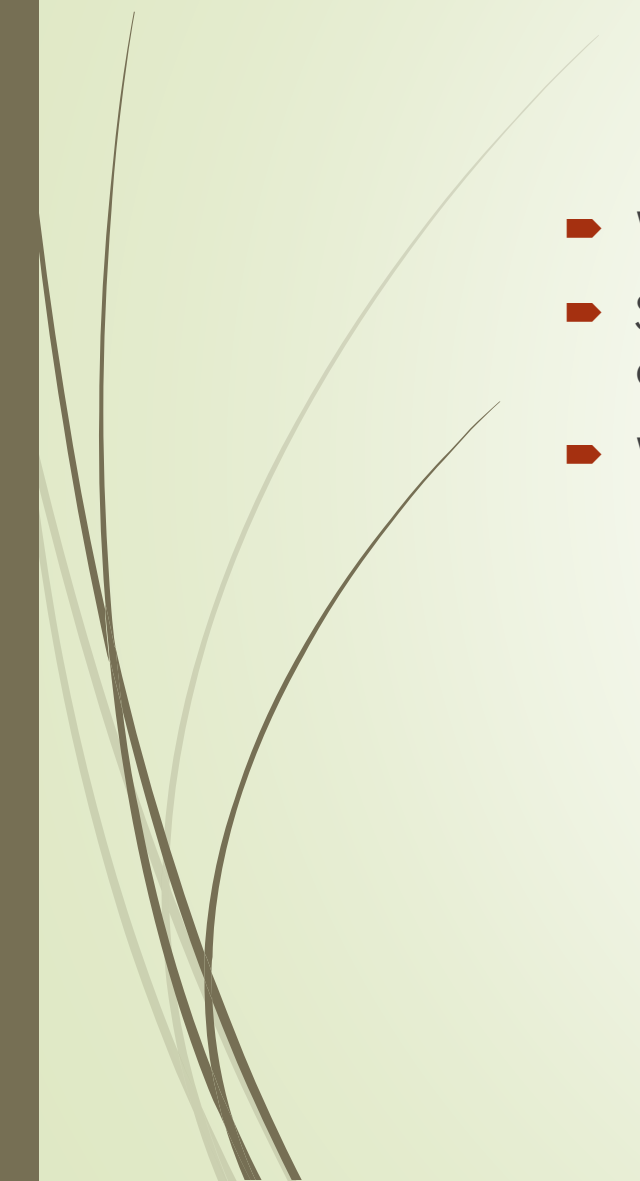
Destructor



- A special type of member function, same name as the class, '~' before the name, no parameter, no return value (**not even void type**).
- Destructor is called when the object is deleted or the scope of the object ends. Cannot explicitly call the destructor.
- Purpose is to free memories and do other tasks when the object is deleted.
- If no destructor is defined, compiler will generate a default destructor, who does nothing.
- If a destructor is defined, no default destructor will be generated.
- A class can only have one destructors.



Destructor

- When the life of an object ends, the destructor is called.
 - Same as constructor for an array of objects, if there are n elements in the array, then constructors and destructors are both called n times.
 - When a dynamic object is deleted by 'delete', the destructor is called.
- 

What Happens?

```
4 class A{
5 public:
6     A(){ cout << "A constructor" << endl; }
7     A(A& a){ cout << "A copy constructor" << endl; }
8     ~A(){ cout << "A destructor" << endl; }
9 };
10 class B{
11 public:
12     B(){ cout << "B constructor" << endl; }
13     B(B& b){ cout << "B copy constructor" << endl; }
14     ~B(){ cout << "B destructor" << endl; }
15 };
16
17 B f(A a){
18     B b;
19     return b;
20 }
21 int main(){
22     A a;
23     f(a);
```

What Happens.

```
4 class A{
5 public:
6     A(){ cout << "A constructor" << endl; }
7     A(A& a){ cout << "A copy constructor" << endl; }
8     ~A(){ cout << "A destructor" << endl; }
9 };
10 class B{
11 public:
12     B(){ cout << "B constructor" << endl; }
13     B(B& b){ cout << "B copy constructor" << endl; }
14     ~B(){ cout << "B destructor" << endl; }
15 };
16
17 B f(A a){
18     B b;
19     return b;
20 }
21 int main(){
22     A a;
23     f(a);
```

```
A constructor
A copy constructor
B constructor
B copy constructor
B destructor
A destructor
B destructor
A destructor
Press any key to continue . . .
```



Static Members

- Use the keyword “static” before defining member variables and functions:
 - `static int a;`
 - `static void f();`
- Usual members have different copies for different objects, while static members only have one copy for the whole class.
- Must initialize static members in **THE SAME FILE** that the class is defined.
- Static member functions can only use static members, including variables and member functions.




Static Member and Static Function

- Ways to access static member:
 - `ObjectName.StaticMemberName`
 - `PointerName->StaticMemberName`
 - `ReferenceName.StaticMemberName`
- Static members can be used by the following, even without any defined object.
 - `ClassName::StaticMemberName`
- `sizeof()` will not count the size of static members.
- Static members are actually global members.
- Example:
 - In a student class, keep a “static int count” to keep track of the number of student objects that have been created.



Order of Constructors and Destructors

- Global variables: constructed when defined (i.e. before main function), destructed when program ends (i.e. after main function).
 - Local variables: constructed when defined, destructed when the scope ends.
 - Static variables: constructed when defined, destructed when the program ends.
- 

What happens?

```
6  class A {
7      int id;
8  public:
9      A(int i) : id(i) { cout << id << " constructor" << endl; }
10     ~A() { cout << id << " destructor" << endl; }
11 };
12
13  A a1(1);
14
15  void f() {
16      cout << "function begins" << endl;
17      A a2(2);
18      static A a3(3);
19      cout << "function ends" << endl;
20  }
21
22  A a4(4);
23
24  int main() {
25      cout << "main begins" << endl;
26      A a5(5);
27      f();
28      cout << "main begins" << endl;
29      return 0;
30  }
```

```

6  class A {
7      int id;
8  public:
9      A(int i) : id(i) { cout << id << " constructor" << endl; }
10     ~A() { cout << id << " destructor" << endl; }
11 };
12
13 A a1(1);
14
15 void f() {
16     cout << "function begins" << endl;
17     A a2(2);
18     static A a3(3);
19     cout << "function ends" << endl;
20 }
21
22 A a4(4);
23
24 int main() {
25     cout << "main begins" << endl;
26     A a5(5);
27     f();
28     cout << "main begins" << endl;
29     return 0;
30 }

```

```

C:\WIN
1 constructor
4 constructor
main begins
5 constructor
function begins
2 constructor
3 constructor
function ends
2 destructor
end begins
5 destructor
3 destructor
4 destructor
1 destructor
Press any key to continue . . .

```

Const variables and pointers

- const variables:
 - `const int a = 1;`
 - `int const a = 1;`
- Pointer to const data:
 - `const int *p;`
- Pointer with const address:
 - `int * const p = &a;`
- Pointer to const data and with const address:
 - `const int * const p = &a;`
- **Cannot assign pointer to const data to ordinary pointer.**



Const references

- ▶ const reference:
 - ▶ `const int &a = b;`
 - ▶ `int const &a = b;`
- ▶ Note: `int & const a = b;` does not make sense, although it can be compiled.

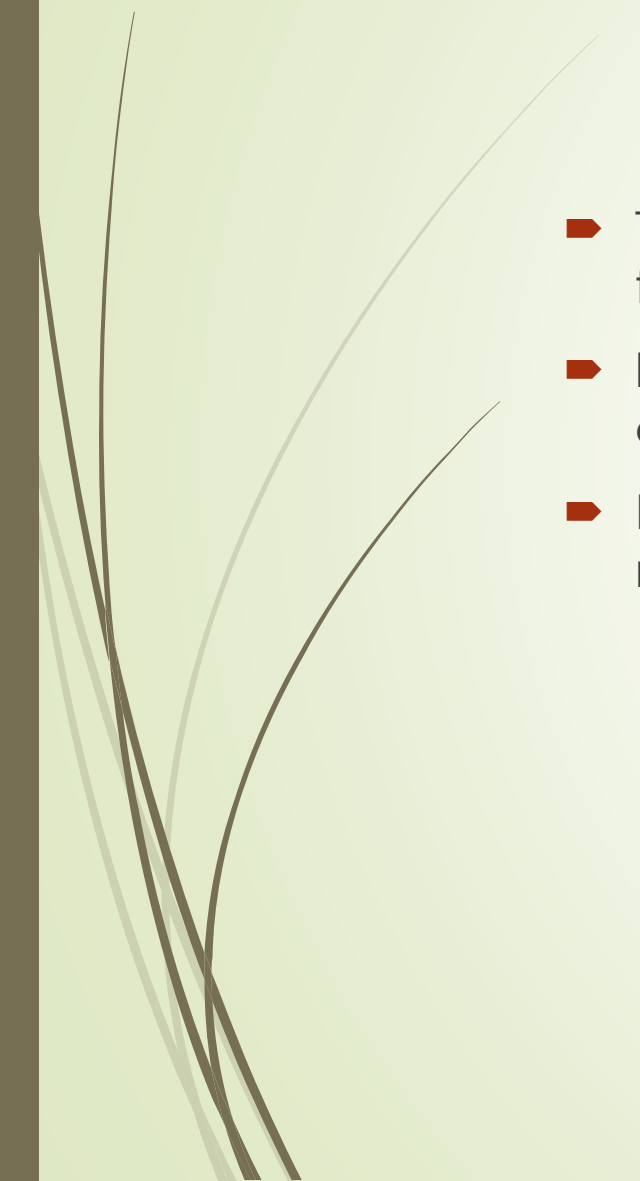


const methods

- “const methods”, a special type of member function:
 - `ReturnType FunctionName(Parameters...) const {...}`
 - This “const” needs to be written when declaring and defining the function.
- Inside const methods:
 - Have access any visible members and other variables outside the class.
 - Cannot modify value of non-constant members of the class.
 - Cannot call non-constant member functions.
 - **Can modify other non-constant variables outside the class.**
- Constant objects can only call constructor, destructor and const methods, but cannot call non-const methods.



const method overloading

- Two member functions, one const method and one not const method, is function overloading.
 - In such case, const objects call const methods and non-constant objects call non-const methods.
 - Remember that two functions that are only different from return types are not overloading.
- 



Enclosing class

- Class with at least one member that is an object of another class.
- Need to initialize these in the parameter list of the constructor.
- When an object of enclosing class is created, constructors of members are called before constructor of the class is called.
- The reverse way for destructor.



Parameter list

- Used in constructor to initialize some members.
- Member objects, const members and references must be initialized in parameter list, or when they are declared.
- The order that constructor of each member is called is based on the order that each member is declared, but is irrelevant with the order that they appear in the parameter list.
- If a member does not appear in the parameter list, it is initialized with default constructor.
- If one initialize member objects, const members and references when they are declared as well as in the parameter list, then the version of parameter list is used.

What happens?

```
4 class A{
5     int id;
6 public:
7     A(int i) :id(i) { cout << id << " constructor" << endl; }
8     ~A() { cout << id << " destructor" << endl; }
9 };
10 class B{
11     A a1;
12     A a2;
13     //const A a3(3); // This is an error.
14     const A a4 = 4;
15     int c;
16     int &d;
17 public:
18     B(int _d) :a2(2), a1(1), a4(5), c(1), d(_d) {}
19 };
20 int main(){
21     int d = 1;
22     B b(d);
23 }
```

```

4 class A{
5     int id;
6 public:
7     A(int i) :id(i) { cout << id << " constructor" << endl; }
8     ~A() { cout << id << " destructor" << endl; }
9 };
10 class B{
11     A a1;
12     A a2;
13     //const A a3(3); // This is an error.
14     const A a4 = 4;
15     int c;
16     int &d;
17 public:
18     B(int _d) :a2(2), a1(1), a4(5), c(1), d(_d) {}
19 };
20 int main(){
21     int d = 1;
22     B b(d);
23 }

```

```

1 constructor
2 constructor
5 constructor
5 destructor
2 destructor
1 destructor

```



friends



- In a class, you can declare some functions, which are not member functions, to be friend of the class. In such case, the function have access to private and protected members.
- That friend function can be member function, constructor or destructor of another class.
- In a class, you can declare another class to be friend of this class. In such situation, the member functions of the friend class can have access to private and protected members of this class.

friends

```
3 class A;
4 class C{
5 public:
6     int add(A);
7 };
8 class A{
9     int a, b;
10 public:
11     A(int _a, int _b) : a(_a), b(_b){ }
12     friend int add(A);
13     friend class B;
14     friend int C::add(A);
15 };
16 int C::add(A a) { return a.a + a.b; }
17 int add(A a){ return a.a + a.b; }
18 class B{
19 public:
20     int add(A a){ return a.a + a.b; }
21 };

23 int main(){
24     A x(1, 2);
25     add(x);
26     B b;
27     b.add(x);
28 }
```

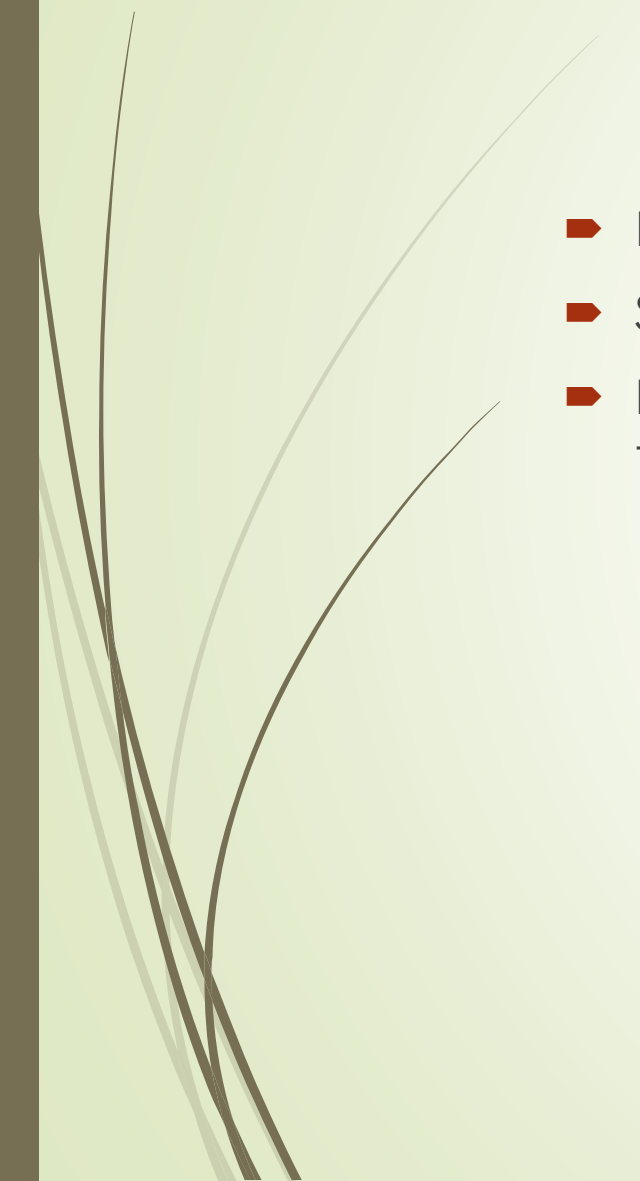
this pointer

- A pointer that points to the object itself.
- Example: can be the return value of a member function.

```
4 class A{  
5     int a;  
6     public:  
7     A* f(){ return this; }  
8     int g(){ return this->a; }  
9 };
```




Operator overloading

- Rewrite operators that have already been defined in C++.
 - Similar as function overloading.
 - Need to be very careful about input type and return type and why we use them.
- 

Overload +, -, *, /

```
4 class complex{
5     double re, im;
6 public:
7     complex() : re(0), im(0) {}
8     complex(double _re) : re(_re), im(0) {}
9     complex(double _re, double _im) : re(_re), im(_im) {}
10
11     complex operator+(const complex& c){ return complex(re + c.re, im + c.im); }
12     //overload as member function.
13     friend complex operator- (const complex&, const complex &);
14     //declare as friend function.
15 };
16 complex operator- (const complex& c1, const complex & c2){
17     return complex(c1.re + c2.re, c1.im + c2.im);
18 }
19 //overload as ordinary function.
```



Overload +, -, *, /

- ▶ If we overload these to be member function, we cannot set class itself to be const, so we have the risk of modify things.
- ▶ In the previous code, what is the true meaning for the following?
 - ▶ $a = b + c;$
 - ▶ $a = b - c;$

Overloading =

➤ How should you overload “=”?

```
323 class complex {  
324     double re, im;  
325 public:  
326     complex() : re(0), im(0) {}  
327     complex(double _re) : re(_re), im(0) {}  
328     complex(double _re, double _im) : re(_re), im(_im) {}
```

Overloading =

```
4 class complex{
5     double re, im;
6 public:
7     complex() : re(0), im(0) {}
8     complex(double _re) : re(_re), im(0) {}
9     complex(double _re, double _im) : re(_re), im(_im) {}
10
11     const complex& operator= (const complex& c){
12         re = c.re;
13         im = c.im;
14         return *this; //this is how you can return the reference
15     }
16 };
17
```



Overloading =

- Use const reference as input.
- Use const reference as output.
- Must be overloaded as member function.
- What does the following mean:
 - $a = b + c;$
 - $a = b = c;$



Why do we overload “=”?

- If we don't overload “=”, it will copy all members directly. This means that the newly created pointer points to the original place and the newly created reference refers to the same variable. This is called **shallow copy**.
- **Deep copy**: the data the pointer points to and variable that the reference refers to are also copied.
- If we overload “=”, we can do deep copy, or a mixture.
- **Lazy copy**: When initially copying an object, the shallow copy is used. A counter is used to track how many objects share the data. When an object is modified, if the counter shows that the data is shared, use a deep copy.



Overloading =

- How would you overload “=” for a class of string?

```
5 class my_string{
6 public:
7     char *str; // This should be private in reality.
8     my_string(){ str = NULL; }
9     const my_string& operator=(const char* _str){
10         if (str == _str)
11             return *this;
12         if (str)
13             delete[] str;
14         str = new char[strlen(_str) + 1];
15         strcpy_s(str, 10, _str);
16         return *this;
17     }
18 };
19 int main(){
20     my_string a;
21     char p[10] = "Cheng Li";
22     a = p;
23     cout << a.str << endl;
24 }
```

Overload << and >> in iostream

➤ How would you overload "<<" and ">>"?

```
323 class complex {  
324     double re, im;  
325     public:  
326     complex() : re(0), im(0) {}  
327     complex(double _re) : re(_re), im(0) {}  
328     complex(double _re, double _im) : re(_re), im(_im) {}
```

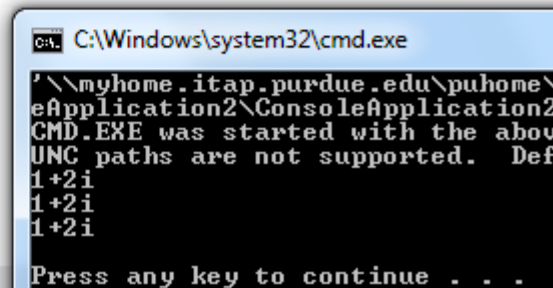


Overload << and >> in iostream

- ▶ ostream is a class and cout is an object of ostream type.
- ▶ You can define another object, say output, and it performs similarly.
- ▶ Example:
 - ▶ ostream output(...);
 - ▶ output << "Cheng Li" << endl;

Overload << and >> in iostream

```
5 class complex{
6     double re, im;
7 public:
8     complex() : re(0), im(0) {}
9     complex(double _re) : re(_re), im(0) {}
10    complex(double _re, double _im) : re(_re), im(_im) {}
11    friend ostream& operator<< (ostream&, complex);
12 };
13
14 ostream& operator<< (ostream& o, complex c){
15     o << c.re << '+' << c.im << 'i' << endl;
16     return o;
17 }
18
19 int main(){
20     complex c(1, 2);
21     cout << c << c << c << endl;
22 }
```



C:\Windows\system32\cmd.exe

```
'\myhome.itap.purdue.edu\puhome\
eApplication2\ConsoleApplication2
CMD.EXE was started with the above
UNC paths are not supported. Def
1+2i
1+2i
1+2i
Press any key to continue . . .
```

Overload << and >> in ostream

- Compare the following:

```
//ostream& operator<< (ostream&, complex);  
//ostream& operator<< (ostream, complex);  
//ostream operator<< (ostream&, complex);  
//ostream operator<< (ostream, complex);  
//void operator<< (ostream&, complex);  
//void operator<< (ostream, complex);
```

Overload << and >> in ostream

```
//ostream& operator<< (ostream&, complex);  
//ostream& operator<< (ostream, complex);  
//ostream operator<< (ostream&, complex);  
//ostream operator<< (ostream, complex);  
//void operator<< (ostream&, complex);  
//void operator<< (ostream, complex);
```

- (2), return reference of a local variable.
- (3) and (4), copy constructor is called unnecessarily.
- (5) and (6), cannot do `cout << a << b << endl;`
- **Note: when overloading ostream, we do not use const reference.**



Overloading []

- How would you overload "[]"?

Overloading []

```
5 class Array{
6     int size;
7     int* p;
8 public:
9     Array(int _size) : size(_size), p(NULL){}
10    ~Array(){ if(p) delete[] p; }
11    int& operator[](int sub){
12        return p[sub];
13    }
14    //We need to return reference since we want to do a[1] = 1;
15};
```




Overloading ++/--

- How would you overload “++/--”?
- 



Overloading ++/--

- ▶ They can be both prefix and postfix.
 - ▶ When overloading as postfix, write an extra parameter, which will not be used.
- 

```
5 class complex{
6     double re, im;
7 public:
8     complex() : re(0), im(0) {}
9     complex(double _re) : re(_re), im(0) {}
10    complex(double _re, double _im) : re(_re), im(_im) {}
11
12    //overload as prefix.
13    complex& operator++(){ re += 1; return *this; }
14
15    //overload as postfix.
16    complex operator++(int){
17        complex temp = *this;
18        re += 1;
19        return temp;
20    }
21
22    friend complex& operator--(complex&);
23    friend complex operator--(complex&, int);
24};
```

```
25 //overload as prefix.
26 complex& operator--(complex& c){
27     c.re -= 1;
28     return c;
29 }
30
31 //overload as postfix.
32 complex operator--(complex& c, int){
33     complex temp = c;
34     c.re -= 1;
35     return temp;
36 }
```



Overloading ++/--

- What does `a++` and `++a` means in each case?
- Pay attention to the parameter type and return type.
- What about `++a++`?

Overloading ()

➡ How would you overload “()”?

```
323 class complex {  
324     double re, im;  
325     public:  
326         complex() : re(0), im(0) {}  
327         complex(double _re) : re(_re), im(0) {}  
328         complex(double _re, double _im) : re(_re), im(_im) {}
```


Overloading ()

- () can be overloaded to do type conversion.

```
5 class complex{  
6     double re, im;  
7     public:  
8         complex() : re(0), im(0) {}  
9         complex(double _re) : re(_re), im(0) {}  
10        complex(double _re, double _im) : re(_re), im(_im) {}  
11  
12        operator double(){ return re; }  
13    };
```




Inheritance

- Already define a class (base class). When defining a new class (derived class), inherit from the existing class.
- Derived class includes all members of base class.
- Can define new members for derived class.
- In derived class, cannot access private member of base class. But can access protected members of base class.
- **Note: protected members cannot be accessed from the outside.**



Inheritance

- Example: a base class for student and a derived class for undergraduate student, and another derived class for graduate student.
- 



Inheritance



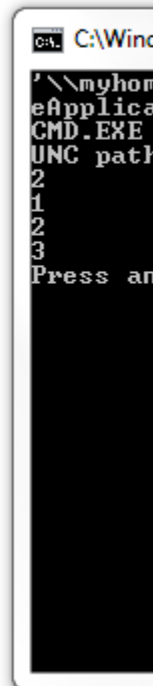
- In derived class, we can define a member that is already defined in base class.
- In such a case, there are two versions, one belonging to the base class and another one belonging to the derived class.
- In such a case, in derived class, by default, the version of derived class is used. In order to access the version in base version, we should use `Base::member...`

Inheritance - Example

```
class A{
public:
    int a = 1;
};

class B : public A{
public:
    int a = 2;
    int b = 3;
};

int main(){
    B b;
    cout << b.a << endl;
    cout << b.A::a << endl;
    cout << b.B::a << endl;
    cout << b.b << endl;
}
```




```
C:\Windows
C:\myhome
eApplica
CMD.EXE
UNC path
2
1
2
3
Press any key to continue...
```



Access Specifier: Protected

- Can be accessed by
 - current class;
 - friend classes;
 - derived classes.



```
class A{
private: int a = 1;
protected: int b = 2;
};

class B : public A{
public:
    int f(){
        cout << a << endl; //error
        cout << b << endl;
    }
};

int main(){
    A a;
    B b;
    a.a; //error
    a.b; //error
    b.a; //error
    b.b; //error
}
```




Three types of inheritance

- Public: all access specifiers from base class remain the same in derived class. (This means that protected members in base class are inherited to be protected members in derived class, which can be accessed by derived class of derived class.)
- Private: all members from base class are inherited to be private members of derived class. (Members inherited from the base class cannot be accessed by derived class of derived class.)
- Protected: all public members are inherited to be protected members in derived class, while all others members possess the same access specifiers.

What are the access specifiers?

```
4 class A {  
5     private: int a;  
6     protected: int b;  
7     public: int c;  
8 };  
9 class B : private A {};  
10 class C : protected A {};  
11 class D : public A {};
```

```
4 class A {
5     private: int a;
6     protected: int b;
7     public: int c;
8 };
9 class B : private A {};
10 class C : protected A {};
11 class D : public A {};
12 class BB : public B { void f() { a; b; c; } }; //a, b, c are private
13 class CC : public C { void f() { a; b; c; } }; //a is private, b, c are protected
14 class DD : public D { void f() { a; b; c; } }; //a is private, b is protected, c is public
15
16 int main() {
17     B b;
18     b.a; //private of B
19     b.b; //private of B
20     b.c; //private of B
21     C c;
22     c.a; //private of C
23     c.b; //protected of C
24     c.c; //protected of C
25     D d;
26     d.a; //private of C
27     d.b; //protected of C
28     d.c; //public of C
29 }
```



Constructor of derived class

- Derived class cannot access private member of base class, but need to initialize them (the base class version).
- Use parameter list and call one of constructors of base class!
- If not, will call the constructor with no parameter of base class.

Constructor of derived class

```
class A{  
private: int a;  
};  
class B : public A{  
public:  
    B(){ a = 1; } //error, cannot access 'a'  
};
```

Constructor of derived class

```
class A{
private: int a;
public: A(int _a) { a = _a; }
};
class B : public A{
public:
    B() : A(1){}
};
class C : public A{
public:
    C() {} //error, no constructor with zero parameter for class A.
};
```


What happens?

```
class A{
public: A() { cout << "Base Constructor" << endl; }
      ~A() { cout << "Base Destructor" << endl; }
};

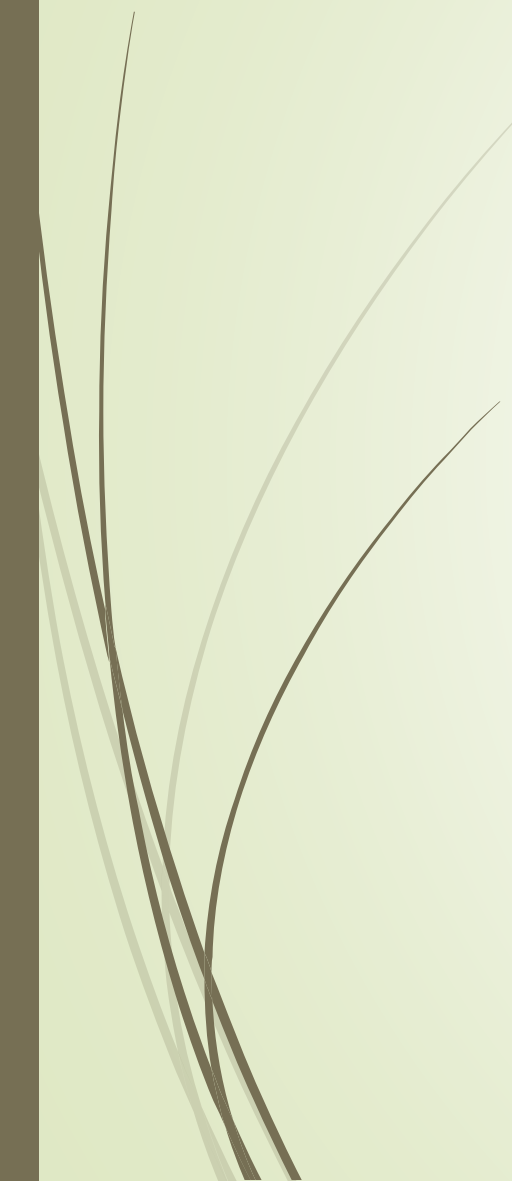
class C{
public: C() { cout << "Member Object Constructor" << endl; }
      ~C() { cout << "Member Object Destructor" << endl; }
};

class B : public A{
    C c;
public: B() { cout << "Derived Constructor" << endl; }
      ~B() { cout << "Derived Destructor" << endl; }
};

int main(){
    B b;
}
```

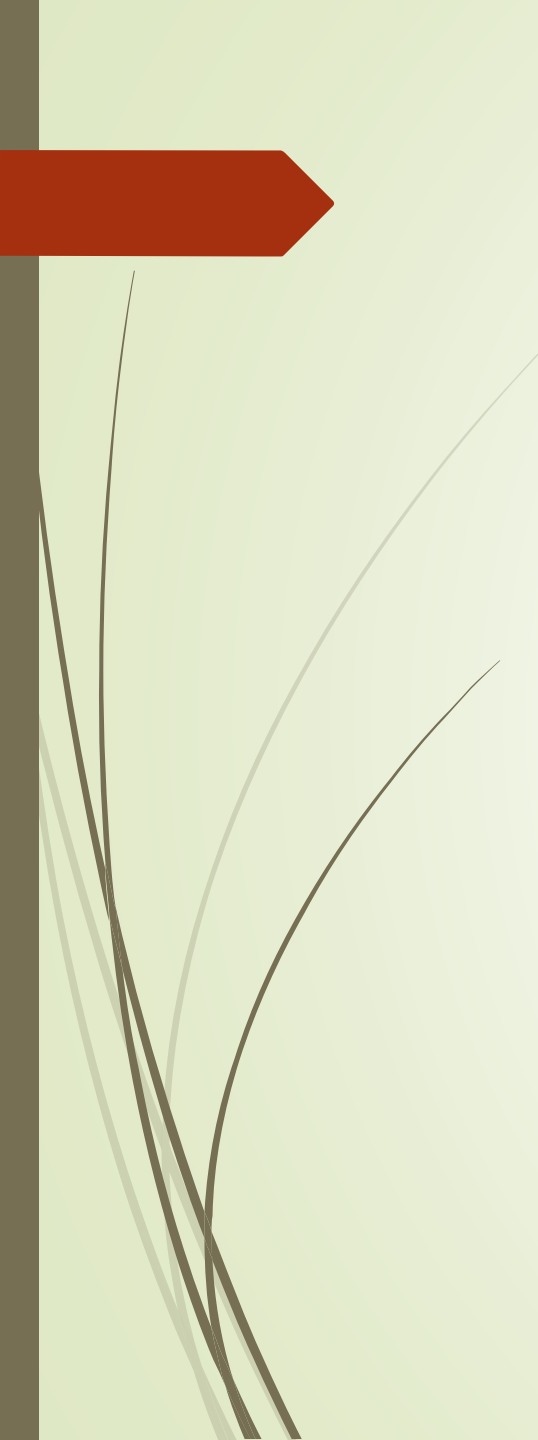
Constructor and destructor of derived class

- Base constructor -> member constructor -> derived constructor
 - Reverse way for destructor.
- 



What happens?





```
1  #include<iostream>
2  using namespace std;
3
4  class C {
5  public: C() { cout << "Derived Member Object Constructor" << endl; }
6         ~C() { cout << "Derived Member Object Destructor" << endl; }
7  };
8
9  class D {
10 public: D() { cout << "Base Member Object Constructor" << endl; }
11         ~D() { cout << "Base Member Object Destructor" << endl; }
12 };
13
14 class A {
15     D d;
16 public: A() { cout << "Base Constructor" << endl; }
17         ~A() { cout << "Base Destructor" << endl; }
18 };
19
20 class B : public A {
21     C c;
22 public: B() { cout << "Derived Constructor" << endl; }
23         ~B() { cout << "Derived Destructor" << endl; }
24 };
25
26 int main() {
27     B b;
28 }
```

```

1  #include<iostream>
2  using namespace std;
3
4  class C {
5  public: C() { cout << "Derived Member Object Constructor" << endl; }
6         ~C() { cout << "Derived Member Object Destructor" << endl; }
7  };
8
9  class D {
10 public: D() { cout << "Base Member Object Constructor" << endl; }
11         ~D() { cout << "Base Member Object Destructor" << endl; }
12 };
13
14 class A {
15     D d;
16 public: A() { cout << "Base Constructor" << endl; }
17         ~A() { cout << "Base Destructor" << endl; }
18 };
19
20 class B : public A {
21     C c;
22 public: B() { cout << "Derived Constructor" << endl; }
23         ~B() { cout << "Derived Destructor" << endl; }
24 };
25
26 int main() {
27     B b;
28 }

```

```

C:\WINDOWS\
Base Member Object Constructor
Base Constructor
Derived Member Object Constructor
Derived Constructor
Derived Destructor
Derived Member Object Destructor
Base Destructor
Base Member Object Destructor
Press any key to continue . . .

```

Type conversion between base and derived class (public inheritance)

```
class A{  
};  
class B : public A{  
};  
  
int main(){  
    A a;  
    B b;  
    a = b; //can assign derived objects to base objects  
    b = a; //error  
    A* p = &b; //can assign derived address to base objects  
    B* q = &a; //error  
    A& x = b; //can assign derived objects to base reference  
    B& y = a; //error  
}
```



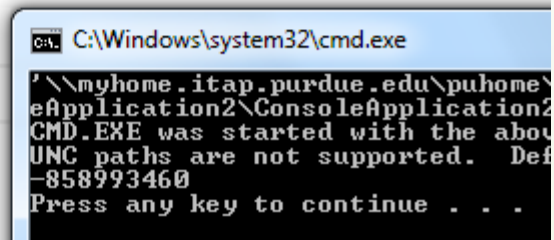
Type conversion between base and derived class (public inheritance)

- If you assign address of derived object to base pointer, can access the members that belong to base class.
- In such a case, the pointer points to a derived object, but since it is of base pointer type, it can only access to members of base class.
- Cannot assign pointer to base class to pointer to derived class.
- Can do type conversion to make a base pointer to be derived pointer, but you need to make sure that the base pointer does point to a derived class.

```
class A{
public: int i = 1;
};


class B : public A{
public: int j = 2;
};

int main(){
    A a;
    B b;
    A* p = &b;
    p->i;
    //p->j; //error, j belongs to derived class.
    B* q = (B*)p; // (B*) does a type conversion.
    q->i;
    q->j; //now can access members of derived class.
    p = &a;
    q = (B*)p;
    cout << q->j << endl;
}
```



cmd C:\Windows\system32\cmd.exe

```
'\\myhome.itap.purdue.edu\puhome\
eApplication2\ConsoleApplication2
CMD.EXE was started with the abou
UNC paths are not supported. Def
-858993460
Press any key to continue . . .
```

Type conversion between base and derived class

- For private and protected inheritance, cannot do these type conversion.
- 



Multiple inheritance

- A class can inherit from several classes.
- The order of constructors:
 - Base classes constructors, according to the order of inheritance
 - Member objects' constructors, according to the order declared
 - Derived class's constructor

What happens?

```
5  class C {  
6      public: C() { cout << "Base 1 Member Object Constructor" << endl; }  
7              ~C() { cout << "Base 1 Member Object Destructor" << endl; }  
8  };  
9  class D {  
10     public: D() { cout << "Base 2 Member Object Constructor" << endl; }  
11             ~D() { cout << "Base 2 Member Object Destructor" << endl; }  
12 };  
13 class B : public C, public D {  
14     C c;  
15     public: B() { cout << "Derived Constructor" << endl; }  
16             ~B() { cout << "Derived Destructor" << endl; }  
17 };  
18  
19 int main() {  
20     B b;  
21 }
```

```

5  class C {
6      public: C() { cout << "Base 1 Member Object Constructor" << endl; }
7              ~C() { cout << "Base 1 Member Object Destructor" << endl; }
8  };
9  class D {
10     public: D() { cout << "Base 2 Member Object Constructor" << endl; }
11             ~D() { cout << "Base 2 Member Object Destructor" << endl; }
12 };
13 class B : public C, public D {
14     C c;
15     public: B() { cout << "Derived Constructor" << endl; }
16             ~B() { cout << "Derived Destructor" << endl; }
17 };
18
19 int main() {
20     B b;
21 }

```

```

C:\WIN
Base 1 Member Object Constructor
Base 2 Member Object Constructor
Base 1 Member Object Constructor
Derived Constructor
Derived Destructor
Base 1 Member Object Destructor
Base 2 Member Object Destructor
Base 1 Member Object Destructor
Press any key to continue . . .

```



Ambiguity from multiple inheritance

- If two base classes both have a variable of the same name, it will create ambiguity.
- But can still access these members by
 - `BaseClassName::variable_name.`
- Ambiguity is checked before access specifier check, so we cannot distinguish two variables of the same name from two base classes by only access specifiers.
- Should be very careful when using multiple inheritance.



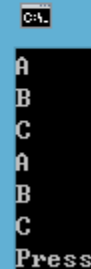
Polymorphism



- Use a pointer to base class, either assigned by address of base class or derived class, to call a virtual function. If the pointer points to the base class, then the virtual function of the base class is called. If otherwise the pointer points to the derived class, then the virtual function of the derived class is called.
- Same for reference.
- Include the keyword “virtual” when declaring the virtual function inside the class, but do not need to include such keyword when writing the function out outside the class.
- When overriding (rewriting) the virtual function in derived class, do not have to include the keyword “virtual”.

Example of polymorphism

```
4 class A {  
5     public: virtual void f() { cout << "A" << endl; }  
6 };  
7 class B : public A {  
8     public: void f() { cout << "B" << endl; }  
9 };  
10 class C : public A {  
11     public: void f() { cout << "C" << endl; }  
12 };  
13 int main() {  
14     A a; B b; C c;  
15     A *p = &a, *q = &b, *r = &c;  
16     p->f(); q->f(); r->f();  
17     A &aa = a, &bb = b, &cc = c;  
18     aa.f(); bb.f(); cc.f();  
19 }
```



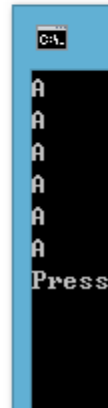
A
B
C
A
B
C
Press

What happens for this?

```
4 class A {  
5     public: void f() { cout << "A" << endl; }  
6 };  
7 class B : public A {  
8     public: void f() { cout << "B" << endl; }  
9 };  
10 class C : public A {  
11     public: void f() { cout << "C" << endl; }  
12 };  
13 int main() {  
14     A a; B b; C c;  
15     A *p = &a, *q = &b, *r = &c;  
16     p->f(); q->f(); r->f();  
17     A &aa = a, &bb = b, &cc = c;  
18     aa.f(); bb.f(); cc.f();  
19 }
```

What happens for this?

```
4  class A {  
5      public: void f() { cout << "A" << endl; }  
6  };  
7  class B : public A {  
8      public: void f() { cout << "B" << endl; }  
9  };  
10 class C : public A {  
11     public: void f() { cout << "C" << endl; }  
12 };  
13 int main() {  
14     A a; B b; C c;  
15     A *p = &a, *q = &b, *r = &c;  
16     p->f(); q->f(); r->f();  
17     A &aa = a, &bb = b, &cc = c;  
18     aa.f(); bb.f(); cc.f();  
19 }
```





Dynamic binding

- It is only until the runtime when it is decided which function to call.
- Use a virtual table to make this possible inside C++.
- All classes with at least one virtual function has a virtual table.
- All objects of such class contain a pointer to the virtual table. Such pointer takes 4 bites to store. The pointer points to the virtual table that contains all virtual functions.
- When a pointer (or reference) to base class calls a virtual function, it goes to the virtual table, and calls the function in the virtual table. In this way, polymorphism is implemented.

What happens?

```
4  class A { void virtual f() {} };
5  class B {};
6  class C { void f() {} };
7  class D { int a; };
8  int main() {
9      cout << sizeof(A) << endl;
10     cout << sizeof(B) << endl;
11     cout << sizeof(C) << endl;
12     cout << sizeof(D) << endl;
13 }
```

What happens

```
4  class A { void virtual f() {} };
5  class B {};
6  class C { void f() {} };
7  class D { int a; };
8  int main() {
9      cout << sizeof(A) << endl;
10     cout << sizeof(B) << endl;
11     cout << sizeof(C) << endl;
12     cout << sizeof(D) << endl;
13 }
```

C++

4

1


1

4

Pre



Access specifier for virtual function

- It depends on the access specifier of the type of the pointer (or reference), but not the type of the object that the pointer points to (or reference refers to).
- 

What happens?

```
4  class A {  
5      private: virtual void f() { cout << "Af" << endl; }  
6      public: virtual void g() { cout << "Ag" << endl; }  
7  };  
8  class B:public A {  
9      public: virtual void f() { cout << "Bf" << endl; }  
10     private: virtual void g() { cout << "Bg" << endl; }  
11 };  
12 int main() {  
13     B b;  
14     A& a = b;  
15     a.f();  
16     a.g();  
17     b.g();
```


What happens

```
4  class A {  
5      private: virtual void f() { cout << "Af" << endl; }  
6      public: virtual void g() { cout << "Ag" << endl; }  
7  };  
8  class B:public A {  
9      public: virtual void f() { cout << "Bf" << endl; }  
10     private: virtual void g() { cout << "Bg" << endl; }  
11 };  
12 int main() {  
13     B b;  
14     A& a = b;  
15     //a.f(); //this is an error, since f is private of A, although public of B  
16     a.g(); // this is OK!  
17     //b.g(); //this is an error, since g is private of B  
18 }
```

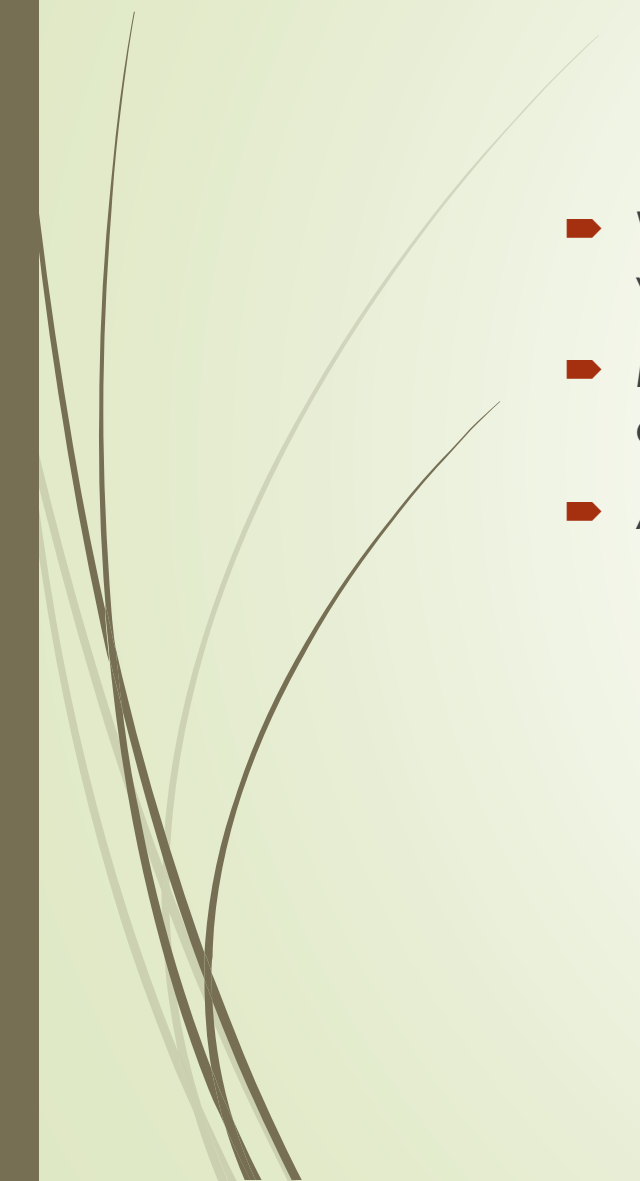


Pure virtual function and abstract class

- Pure virtual function: a virtual function without the body of function.
- Syntax: `virtual Return Type FunctionName (...) = 0;`
- Abstract class: a class containing pure virtual function.
- Cannot define object of abstract class.
- Abstract class is only used for inheritance.
- But can define pointer or reference to abstract class, and assign them by address or object of derived class.
- In a derived class of an abstract class, if all pure virtual functions are overridden, then the class is not an abstract class. Otherwise, it is still an abstract class.



Pure virtual function and abstract class

- ▶ When constructors and destructor call a virtual function, they call the version of the current class, but not derived class.
 - ▶ Member functions of abstract class can call pure virtual functions, but constructors and destructor cannot call pure virtual function.
 - ▶ Abstract class needs to have constructors and destructor.
- 

Example

```
5 class A {  
6 public:  
7     virtual void f() = 0;  
8     A() { f(); } //This is an error.  
9 };  
10 class B : public A {  
11 public:  
12     void f() {}  
13 };  
14 int main() {  
15     B b;  
16 }
```

```
5 class A {  
6 public:  
7     virtual void f() {};  
8     A() { f(); } //This is OK.  
9 };  
10 class B : public A {  
11 public:  
12     void f() {}  
13 };  
14 int main() {  
15     B b;  
16 }
```



Virtual destructor

- ▶ When deleting an object through a pointer (or reference) to a base class, may not be able to delete everything in derived class.
- ▶ Generally, we should call the base class destructor and then call the derived class destructor, but a pointer (or reference) to base class will not do this.
- ▶ To solve the problem, use virtual destructor.
- ▶ For derived class, no need to declare virtual to destructor, as long as virtual destructor is declared in base class.
- ▶ Generally, if a class has a virtual function, then we should use virtual destructor.

Example

```
5  class A {  
6      int* p;  
7  public:  
8      virtual ~A() { delete[] p; }  
9  };  
10 class B : public A {  
11     int* q;  
12 public:  
13     ~B() { delete[] q; }  
14 };  
15 int main() {  
16     B b;  
17 }
```



Template function

- We can use define classes or basic data types as template parameters to define a group of functions.
- Template parameters can either be used as input or return types of the function or be used inside the function.


```

5   template<class T1, class T2>
6   void f(T1 a, T2 b) {
7       cout << sizeof(a) << endl;
8       cout << sizeof(b) << endl;
9   }
10
11  template<class T1, class T2, class T3>
12  void g(T1 a, T2 b) {
13      cout << sizeof(a) << endl;
14      cout << sizeof(b) << endl;
15  }
16
17  template<int x, bool y, class T> //double cannot be template parameter
18  void h(T a, T b) {
19      cout << x << "\t" << y << endl;
20  }
21
22  int main() {
23      f(1.0, 2);
24      f(true, 0);
25      g<int, double, void>(1, 2);
26      g<char, bool, void>(1, 2);
27      h<1, true, int>(1, 2);
28      return 0;
29  }

```

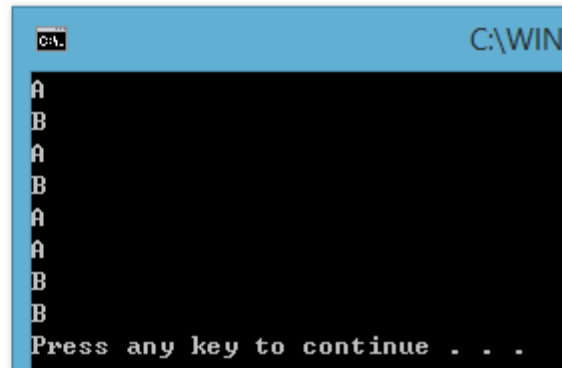
```

C:\WINDOWS\sy
8
4
1
4
4
8
1
1
1
1
1
Press any key to continue . . .

```

Example

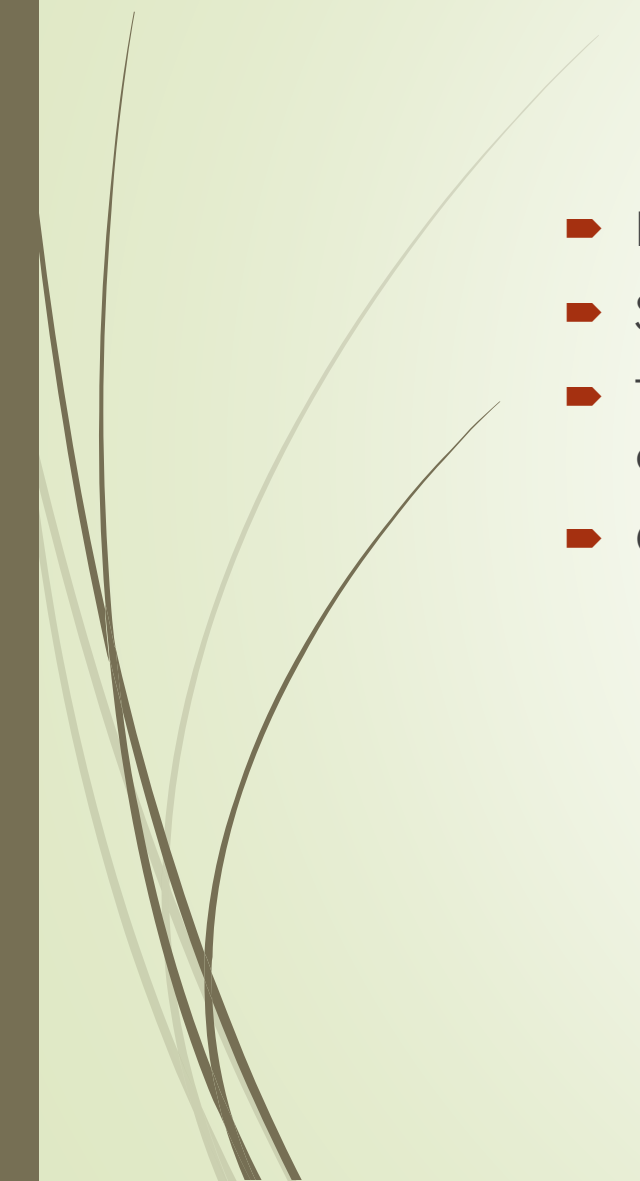
```
4 class A { public: void g() { cout << "A" << endl; } };
5 class B { public: void g() { cout << "B" << endl; } };
6 template<class T1, class T2>
7 void f(T1 x, T2 y) { x.g(); y.g(); }
8 int main() {
9     A a; B b;
10    f<A, B>(a, b);
11    f(a, b);
12    f<A, A>(a, a);
13    f(b, b);
14 }
```



```
C:\WIN
A
B
A
B
A
A
A
B
B
Press any key to continue . . .
```



Which function to call?

- First, choose a function with exactly the same parameters.
 - Second, choose a template function with exactly the same parameters.
 - Third, without ambiguity, find a function that, after type conversion, have appropriate parameter types.
 - Otherwise, an error occurs.
- 



Template and class

- ▶ We can use define classes or basic data types as template parameters to define a group of classes.
- ▶ Template parameters can either be used as for member variables and functions.
- ▶ All other mechanism, like inheritance, polymorphism can be used similarly.



STL: Standard Template Library

- Two key features of C++:
 - (1) object oriented programming: class, inheritance, polymorphism, etc.
 - (2) generic programming: template.
- STL includes a large class of data structures, implemented by template. Algorithms can be developed without specifying the data structure.
 - For example, if a class of data structures has the so-called random access iterator, then a sorting algorithm can be developed for all the data structures.
- With STL, no need to write most common data structures and algorithms, and they also have better performance.
- STL was added to C++ in 1998.



STL Components



- Containers: data structure containing different types of data with different structures, e.g. vector, stack, queue, etc.
- Iterators: access elements in containers with specific rules.
- Algorithms: algorithms for containers, e.g. sorting for vector.

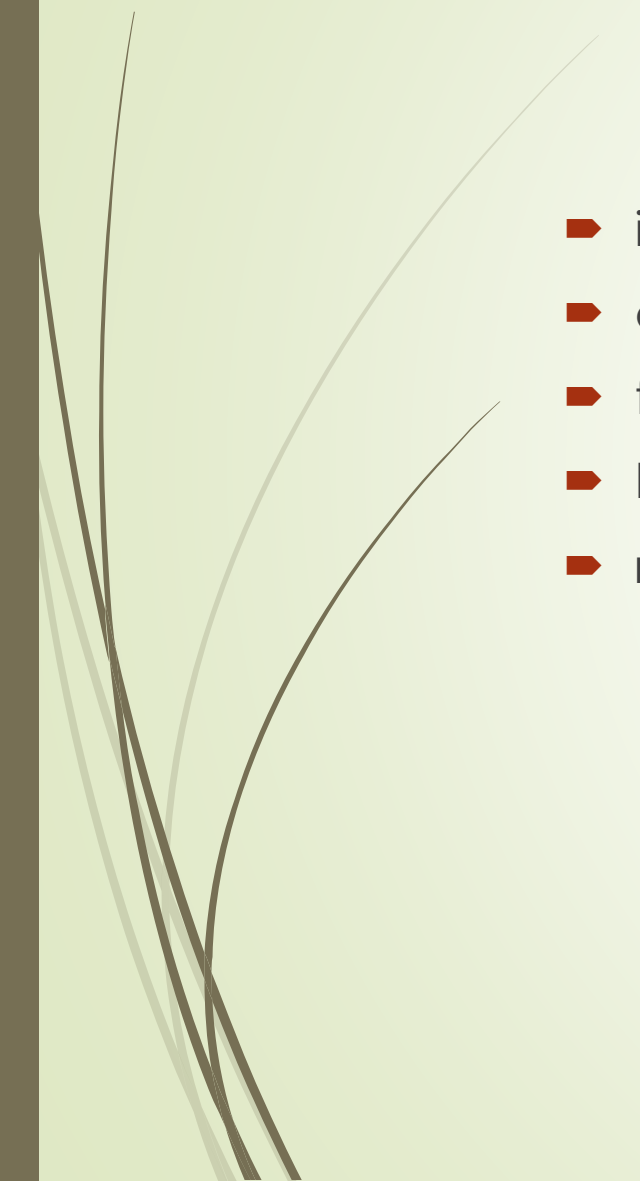


Containers

- Sequences: vector, list, slist, deque.
- Container adapter: queue, priority_queue, stack.
- Associative containers: set, multiset, map, multimap.
- Type one container: sequences and associative containers.



Iterator

- input iterators: read only.
 - output iterators: write only.
 - forward iterators: read, written to and move forward.
 - bidirectional iterators: read, written to and move forward and backwards.
 - random access iterators: read and write freely.
- 



Iterator

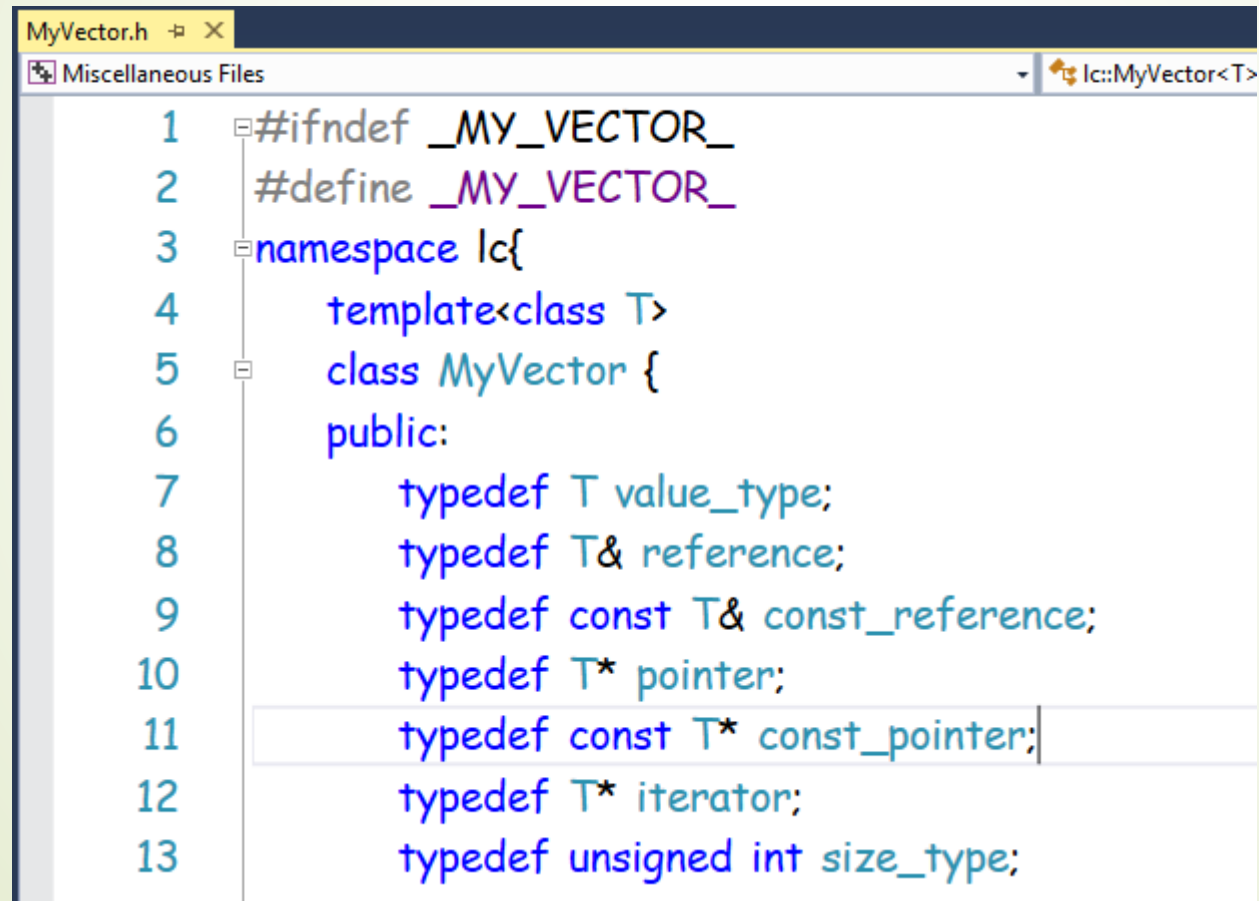
- For selection, insertion, bubble, quick, merge, heap sort, which iterator can support them?
- What kind of iterator can be supported for linked list and array?



Example: *MyVector*

- A similar template class as vector in STL.
- 

Some type defines



The image shows a code editor window titled 'MyVector.h'. The editor displays the following C++ code:

```
1  #ifndef _MY_VECTOR_
2  #define _MY_VECTOR_
3  namespace lc{
4      template<class T>
5      class MyVector {
6      public:
7          typedef T value_type;
8          typedef T& reference;
9          typedef const T& const_reference;
10         typedef T* pointer;
11         typedef const T* const_pointer;
12         typedef T* iterator;
13         typedef unsigned int size_type;
```

The code defines a namespace 'lc' containing a template class 'MyVector'. Inside the class, several typedefs are used to define standard container types: 'value_type' (T), 'reference' (T&), 'const_reference' (const T&), 'pointer' (T*), 'const_pointer' (const T*), 'iterator' (T*), and 'size_type' (unsigned int).



Private members

```
15     private:  
16         T* p;  
17         size_type n;  
18         size_type _reserved_size;
```

Constructors and destructor and “=”

```
20     public:
21         //constructor and destructor
22         //Explicit constructors are simply constructors
23         //that cannot take part in an implicit conversion.
24         //Explicit should only be declared inside the class.
25         explicit MyVector();
26         explicit MyVector(size_type);
27         explicit MyVector(size_type, const_reference);
28         MyVector(iterator, iterator);
29         MyVector(MyVector<T>&);
30         MyVector& operator=(MyVector&);
31         ~MyVector() { if(p != NULL) delete[] p; }
```

Some simple public members

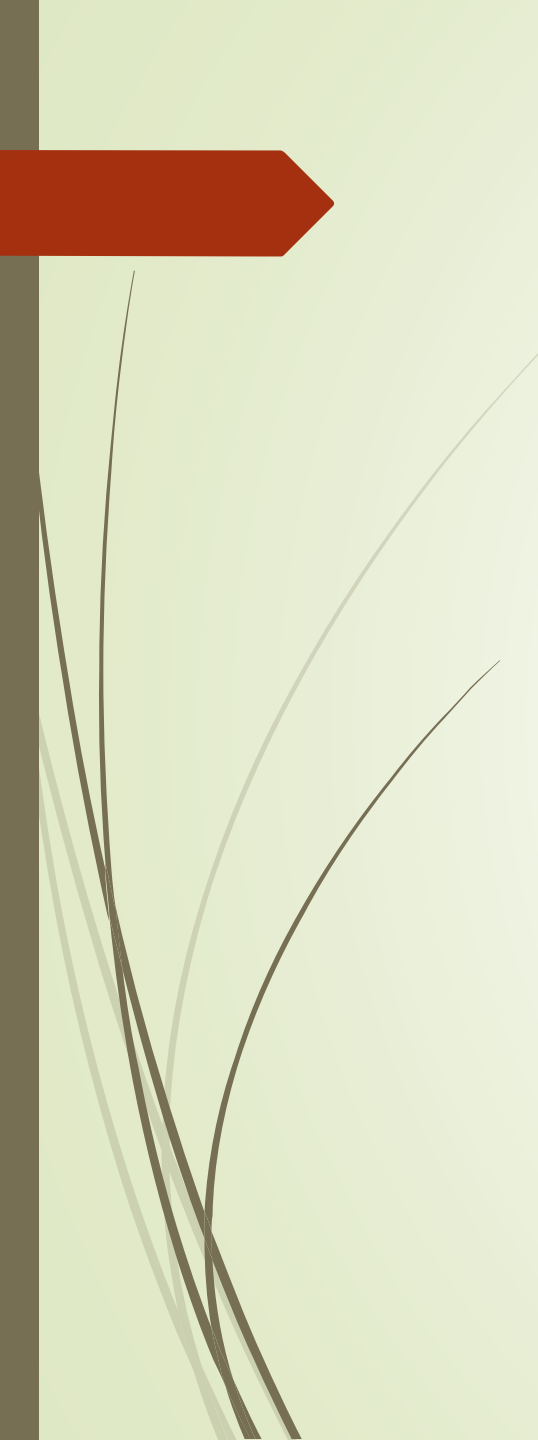
```
33 //iterator
34 iterator begin() {return p;}
35 iterator end() {return p + n;}
36
37 size_type size() const { return n; }
38 size_type capacity() const { return _reserved_size; }
39 void reserve(int);
40 bool empty() const { return n == 0; }
41
42 reference operator[](int i) { return p[i]; }
43 reference front() { return p[0]; }
44 reference back() { return p[n - 1]; }
```


Modification public methods


```
46     void assign(iterator, iterator);
47     void assign(size_type, const_reference);
48     void push_back(const_reference);
49     void pop_back();
50     void insert(iterator, const_reference);
51     void insert(iterator, size_type, const_reference);
52     void erase(iterator);
53     void clear() { n = 0; }
54 };
```

Constructors

```
56     template<class T>
57     MyVector<T>::MyVector() {
58         p = new value_type[1];
59         n = 0;
60         _reserved_size = 1;
61     }
62     template<class T>
63     MyVector<T>::MyVector(size_type _n) {
64         p = new value_type[_n];
65         n = _reserved_size = _n;
66     }
67     template<class T>
68     MyVector<T>::MyVector(size_type _n, const_reference val) {
69         p = new value_type[_n];
70         for (int i = 0; i < _n; ++i)
71             p[i] = val;
72         n = _reserved_size = _n;
73     }
```



```
75     template<class T>
76     MyVector<T>::MyVector(iterator first, iterator last) {
77         n = last - first;
78         if (n > _reserved_size) {
79             if(p != nullptr)
80                 delete[] p;
81             p = new value_type[n];
82             _reserved_size = n;
83         }
84         iterator it = first;
85         while (it != last) {
86             p[it - first] = *it;
87             ++it;
88         }
89     }
90     template<class T>
91     MyVector<T>::MyVector(MyVector<T>& x) {
92         _reserved_size = n = x.size();
93         p = new value_type [n];
94         for (int i = 0; i < n; ++i)
95             p[i] = x[i];
96     }
```



```
97     template<class T>
98     MyVector<T>& MyVector<T>::operator=(MyVector<T>& x) {
99         n = x.size();
100        if (n > _reserved_size) {
101            if(p != nullptr)
102                delete[] p;
103            p = new value_type[n];
104            _reserved_size = n;
105        }
106        for (int i = 0; i < n; ++i)
107            p[i] = x[i];
108        return *this;
109    }
```

Reserve more space

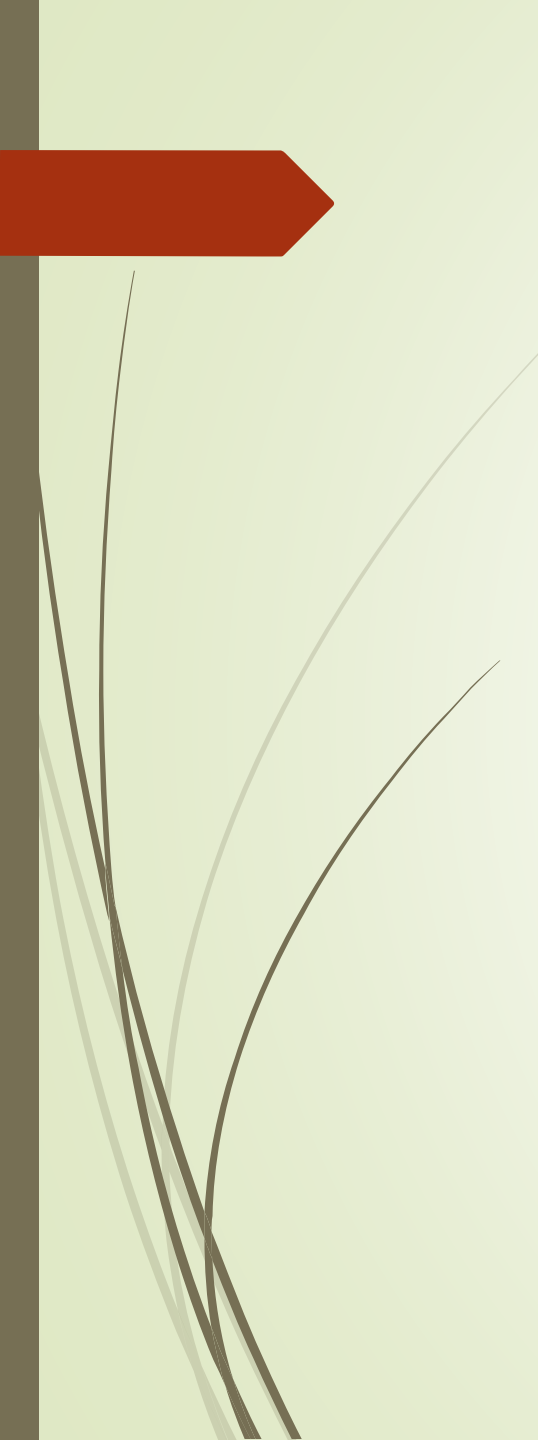
```
121     template<class T>
122     void MyVector<T>::reserve(int _n){
123         if(n >= _n)
124             return;
125         pointer np = new value_type[_n];
126         for(int i = 0; i < n; ++i)
127             np[i] = p[i];
128         pointer to_delete = p;
129         p = np;
130         if(to_delete != nullptr)
131             delete to_delete;
132         _reserved_size = _n;
133
134     }
```

Assign the vector


```
135     template<class T>
136     void MyVector<T>::assign(iterator first, iterator last){
137         size_type _n = last - first;
138         if(n < _n)
139             reserve(_n);
140         for(int i = 0; i < _n; ++i)
141             p[i] = *(first + i);
142         return;
143     }
144     template<class T>
145     void MyVector<T>::assign(size_type _n, const_reference val){
146         if(n < _n)
147             reserve(_n);
148         for(int i = 0; i < _n; ++i)
149             p[i] = val;
150         return;
151     }
```

Modify the vector

```
154     template<class T>
155     void MyVector<T>::push_back(const_reference x) {
156         if(n == _reserved_size){
157             reserve(2 * n + 1);
158             _reserved_size = 2 * n + 1;
159         }
160         p[n++] = x;
161         return;
162     }
163     template<class T>
164     void MyVector<T>::pop_back() {
165         if (n == 0)
166             return;
167         --n;
168         return;
169     }
```

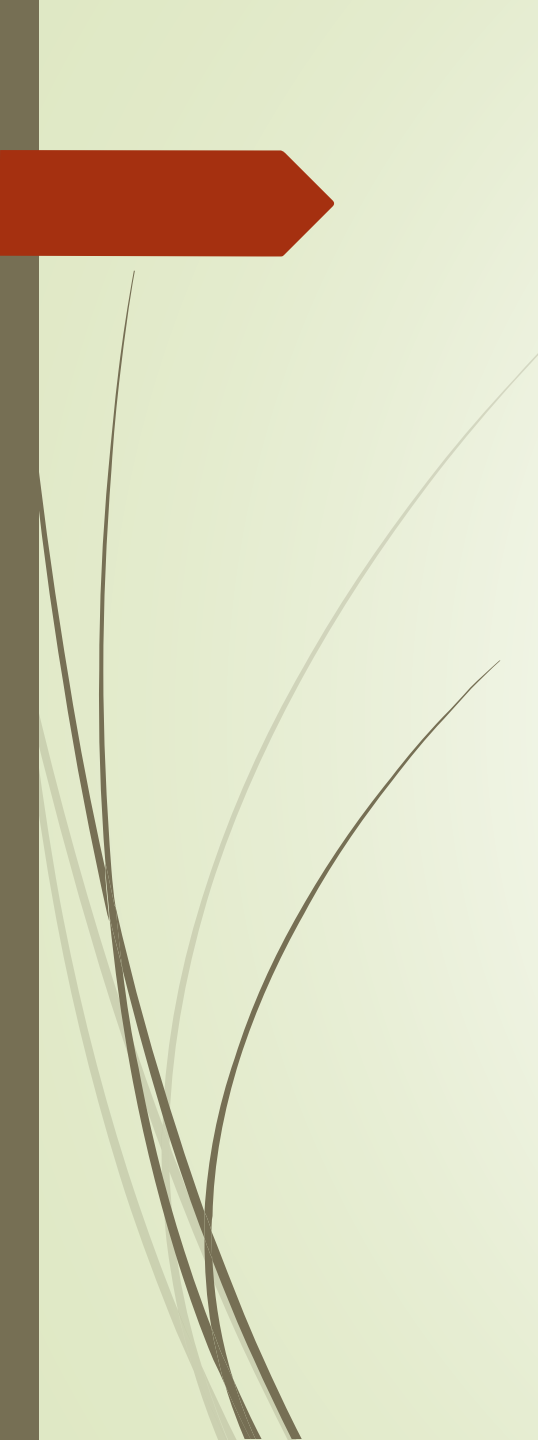
```
170     template<class T>
171     void MyVector<T>::insert(iterator pos, const_reference val) {
172         insert(pos, 1, val);
173     }
174     template<class T>
175     void MyVector<T>::insert(iterator pos, size_type _n, const_reference val) {
176         int i = pos - p;
177         if (i < 0 || i > n)
178             return;
179         if(n + _n > _reserved_size){
180             reserve(n + _n);
181             _reserved_size = n + _n;
182         }
183         int k = n - 1;
184         for(; k >= i; --k)
185             p[k + _n] = p[k];
186         k += _n;
187         for(; k >= i; --k)
188             p[k] = val;
189         n += _n;
190         return;
191     }
```



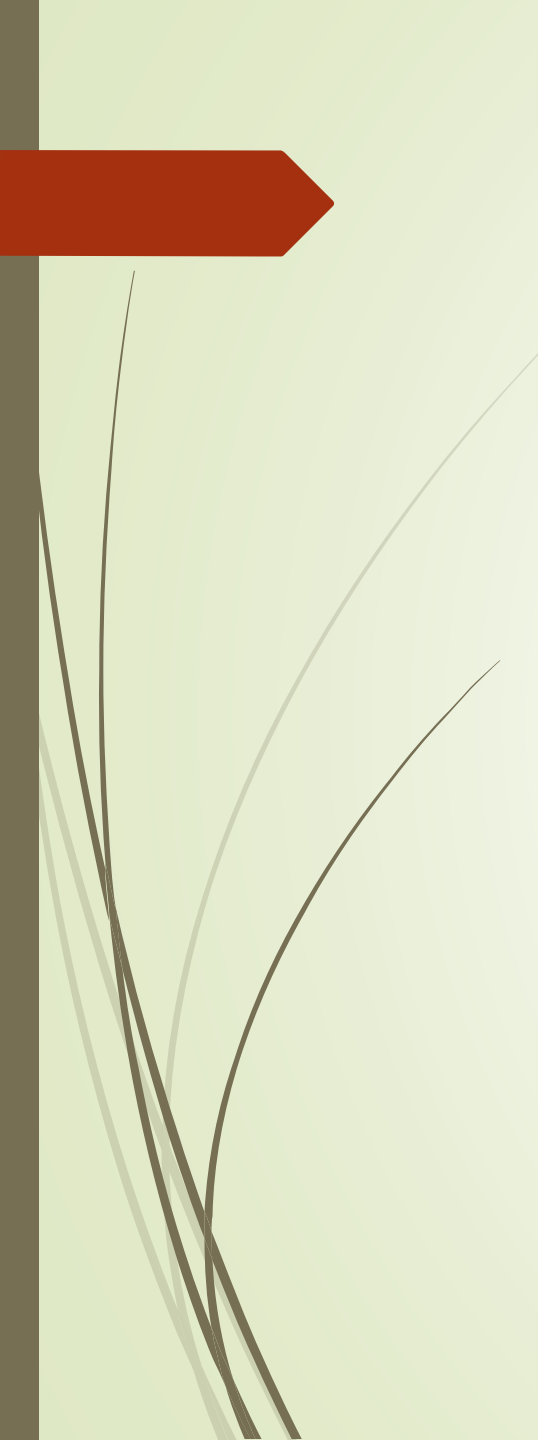
```
192     template<class T>
193     void MyVector<T>::erase(iterator pos) {
194         int i = pos - p;
195         if (i < 0 || i >= n)
196             return;
197         while(i != n - 1) {
198             p[i] = p[i + 1];
199             ++i;
200         }
201         --n;
202         return;
203     }
```

Some testing

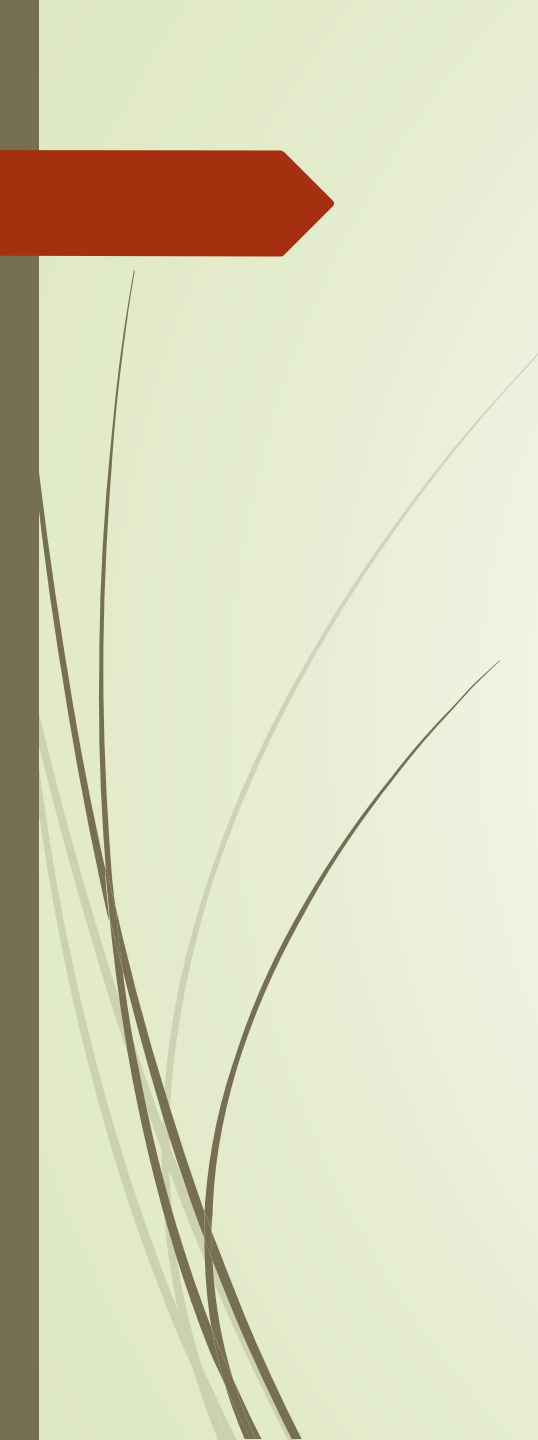
```
1  #include<iostream>
2  #include "MyVector.h"
3  using namespace lc;
4
5  int main() {
6      std::cout << "Part 1: Constructors." << std::endl;
7      MyVector<int> test;
8      print(test);
9      MyVector<int> test1(5);
10     print(test1);
11     MyVector<int> test2(5, 1);
12     print(test2);
13     MyVector<int> test3(test2.begin(), test2.end());
14     print(test3);
15     MyVector<int> test4(test3);
16     print(test4);
17     test4 = test1;
18     print(test4);
```



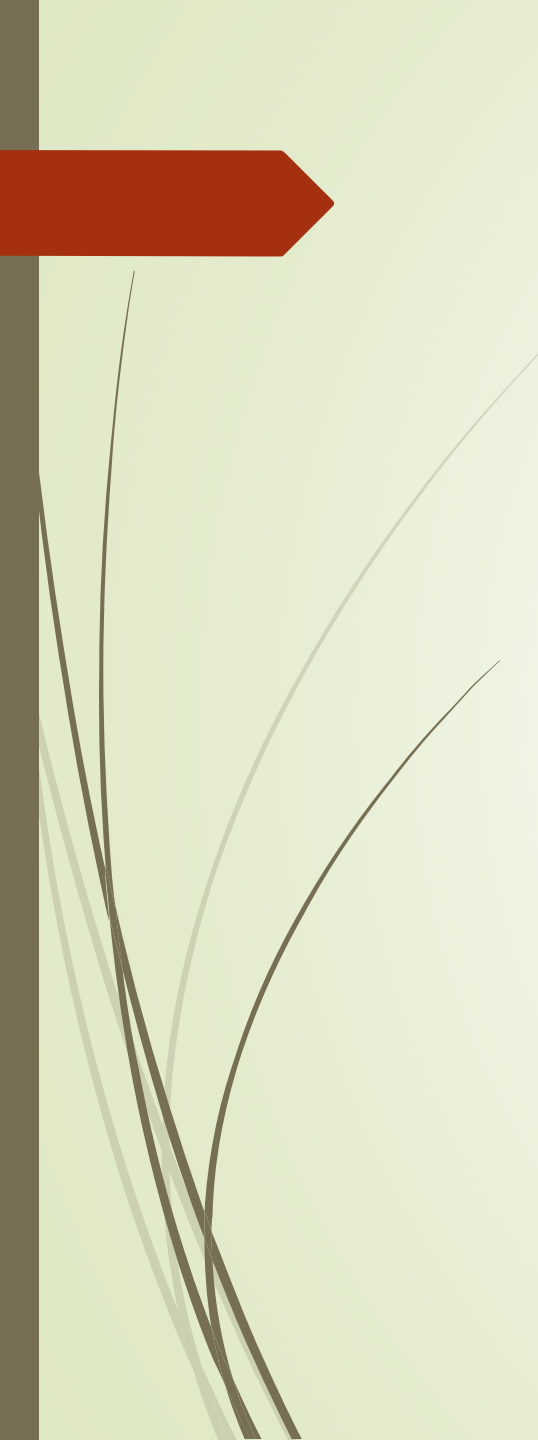
```
20 std::cout << std::endl;
21 std::cout << "Part 2: Sizes." << std::endl;
22 std::cout << "Current vector: ";
23 print(test);
24 std::cout << "Is empty? " << test.empty() << std::endl;
25 std::cout << "Size: " << test.size() << "." << std::endl;
26 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
27
28 test.push_back(0);
29 std::cout << "Current vector: ";
30 print(test);
31 std::cout << "Is empty? " << test.empty() << std::endl;
32 std::cout << "Size: " << test.size() << "." << std::endl;
33 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
34
35 test.push_back(1);
36 std::cout << "Current vector: ";
37 print(test);
38 std::cout << "Size: " << test.size() << "." << std::endl;
39 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
```



```
41 test.push_back(2);
42 std::cout << "Current vector: ";
43 print(test);
44 std::cout << "Size: " << test.size() << "." << std::endl;
45 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
46
47 test.push_back(3);
48 std::cout << "Current vector: ";
49 print(test);
50 std::cout << "Size: " << test.size() << "." << std::endl;
51 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
52
53 test.push_back(4);
54 std::cout << "Current vector: ";
55 print(test);
56 std::cout << "Size: " << test.size() << "." << std::endl;
57 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
```



```
59 test.reserve(10);
60 std::cout << "Current vector: ";
61 print(test);
62 std::cout << "Size: " << test.size() << "." << std::endl;
63 std::cout << "Capacity: " << test.capacity() << "." << std::endl;
64
65 std::cout << std::endl;
66 std::cout << "Part 3: Elements access." << std::endl;
67 test1 = test;
68 test1[1] = test1[2] = test1[3] = 2;
69 test1.front() = test1.back() = 1;
70 print(test1);
71
72 std::cout << "Part 4: Modifiers." << std::endl;
73 test1.assign(5, 2);
74 print(test1);
75 test1.assign(test.begin(), test.end());
76 print(test1);
```

```
78 test1.pop_back();
79 print(test1);
80 test1.pop_back();
81 print(test1);
82
83 test1.insert(test1.begin(), 100);
84 print(test1);
85 test1.insert(test1.begin() + 2, 2, 200);
86 print(test1);
87
88 test1.erase(test1.begin() + 2);
89 print(test1);
90 test1.erase(test1.begin() + 2);
91 print(test1);
92 test1.erase(test1.begin());
93 print(test1);
94
95 test1.clear();
96 print(test1);
```


Part 1: Constructors.

-842150451 -842150451 -842150451 -842150451 -842150451
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
-842150451 -842150451 -842150451 -842150451 -842150451

Part 2: Sizes.

Current vector:

Is empty? 1

Size: 0.

Capacity: 1.

Current vector: 0

Is empty? 0

Size: 1.

Capacity: 1.

Current vector: 0 1

Size: 2.

Capacity: 3.

Current vector: 0 1 2

Size: 3.

Capacity: 3.

Current vector: 0 1 2 3

Size: 4.

Capacity: 7.

Current vector: 0 1 2 3 4

Size: 5.

Capacity: 7.

Current vector: 0 1 2 3 4

Size: 5.

Capacity: 10.

Part 3: Elements access.

1 2 2 2 1

Part 4: Modifiers.

2 2 2 2 2

0 1 2 3 4

0 1 2 3

0 1 2

100 0 1 2

100 0 200 200 1 2

100 0 200 1 2

100 0 1 2

0 1 2

Press any key to continue . . .



Another Example: Matrix



```

template<class T, int m, int n>
class Matrix {
    vector<vector<T>> elements;
    int nrow;
    int ncol;
public:
    Matrix();
    Matrix(Matrix<T, m, n>&);

    T& operator() (int, int); //get elements in the matrix
    vector<T> row(int); //get ith row as a vector
    vector<T> col(int); //get jth column as a vector

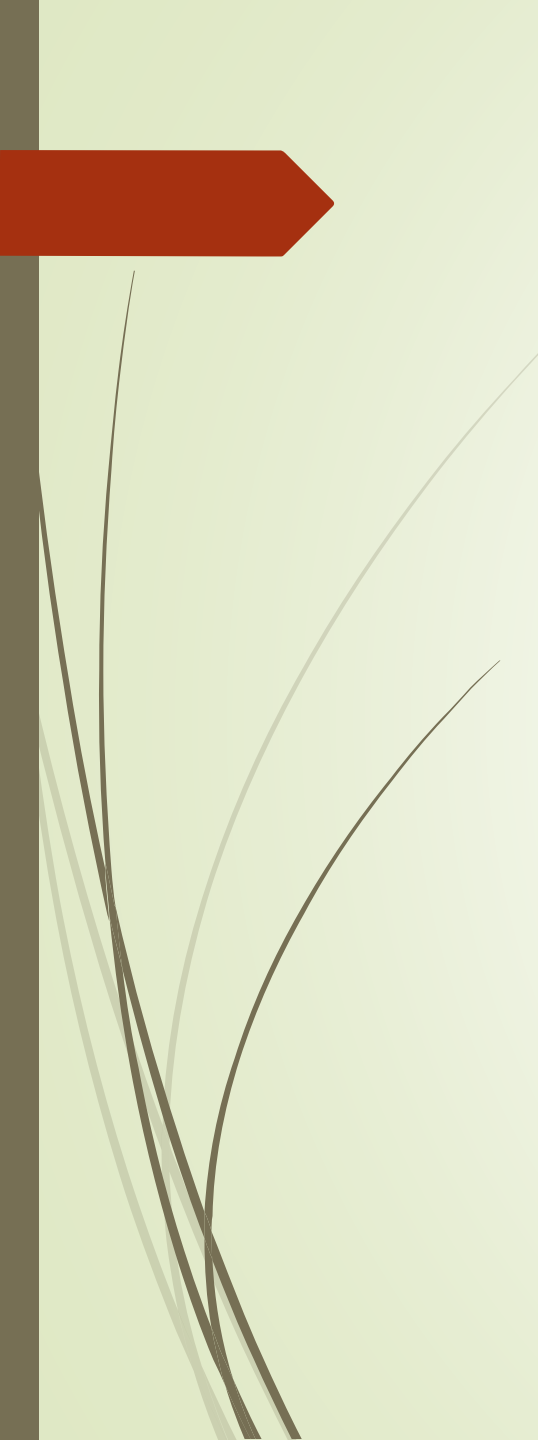
    //need to overload +, - and *
    //For *, consider both scalar and matrix multiplication

    Matrix<T, n, m> transpose(); //return the transpose of the matrix
    Matrix<T, m, n> GaussElimination(); //return the Gauss elimination
    int rank(); //return the rank of the matrix
    vector<T> Solve(const vector<T>&); //solve a linear equation
};

template<class T, int m, int n>
Matrix<T, m, n>::Matrix() () {...}

Matrix<int, 2, 2> a;

```



```
template<class T, int n>
class SquareMatrix :public Matrix<T, n, n> {
public:
    SquareMatrix();
    SquareMatrix(SquareMatrix<T, n>&);

    bool isNonSingular(); //return if the matrix is singular
    SquareMatrix<T, n> inverse(); //return the inverse of the matrix

    SquareMatrix<T, n> pow(int n); //calculate A^n

    vector<T> Eigenvalue(); //return all the eigenvalues, in descending order, with multiplication
    vector<T> Eigenvector(T); //return eigenvector corresponding to a certain eigenvalue

    SquareMatrix<T, n> cholesky(); //return the Cholesky decomposition

    SquareMatrix<T, n> Q(); //return the Q of QR decomposition
    SquareMatrix<T, n> R(); //return the R of QR decomposition
};
```