



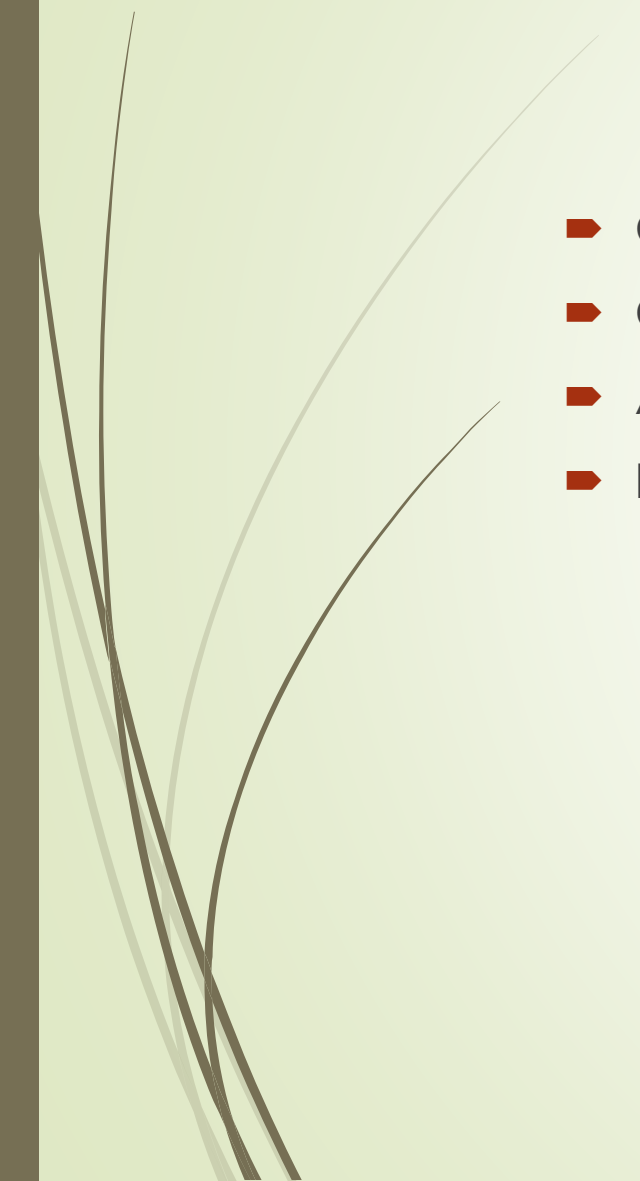
# STAT 598W Object Oriented Programming in C++

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.

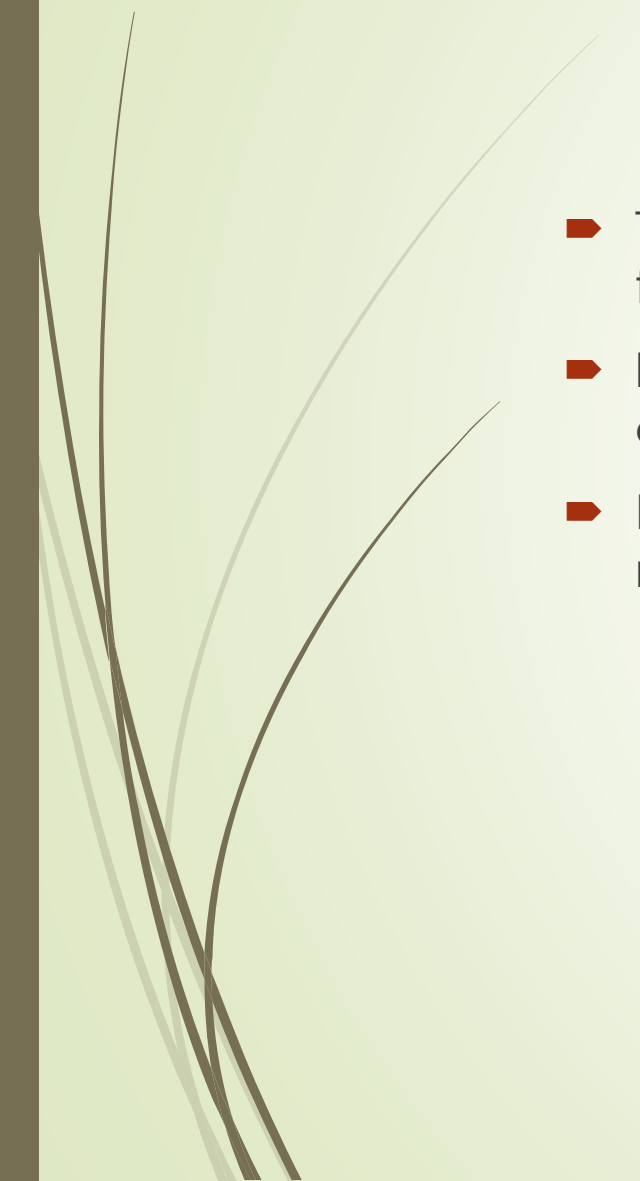


# const methods

- “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

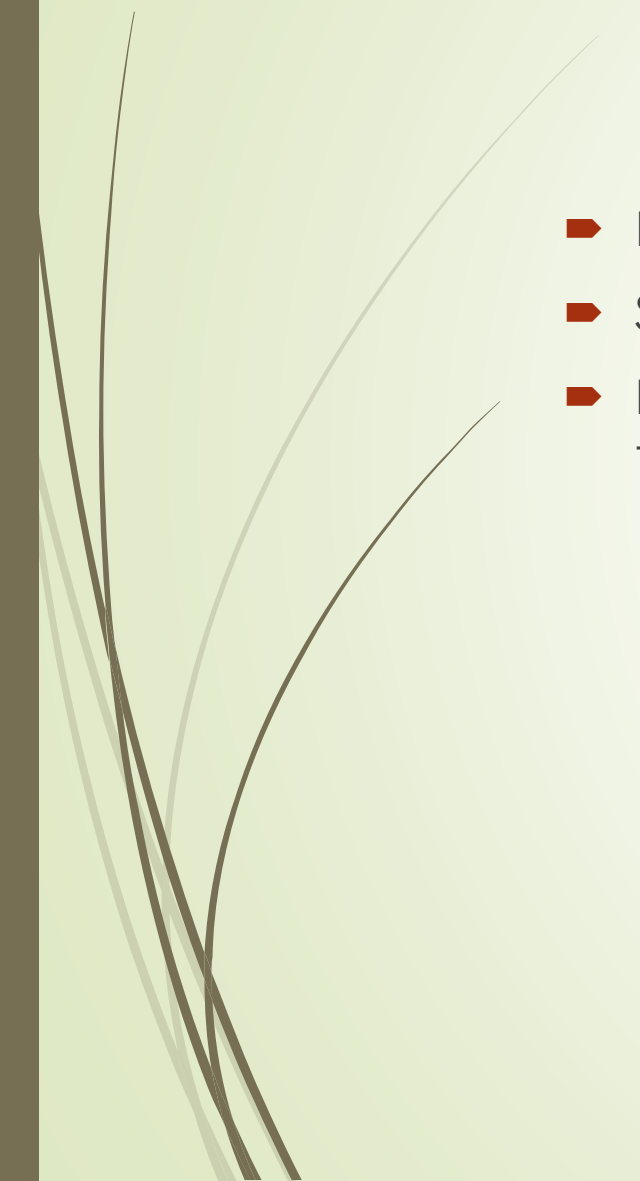
- A pointer that points to the object itself.
- Example: can be the return value of a member function.

```
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 “=”?

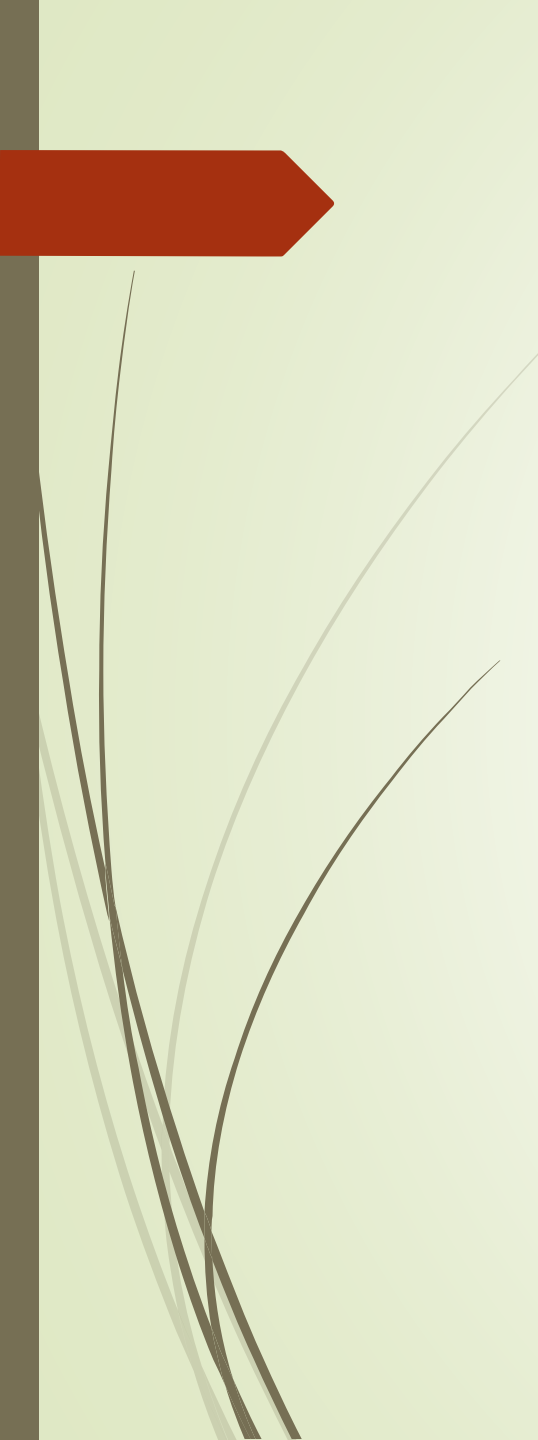
- If we don't overload “=”, it will copy all members directly. This means that the newly created pointer points to the original place and the newly created reference refers to the same variable. This is called **shallow copy**.
- **Deep copy**: the data the pointer points to and variable that the reference 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) {}
```

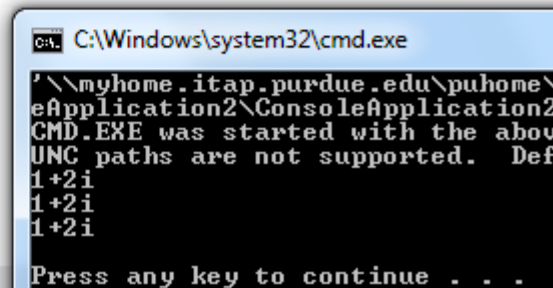


# Overload << and >> in iostream

- ▶ 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    };
```