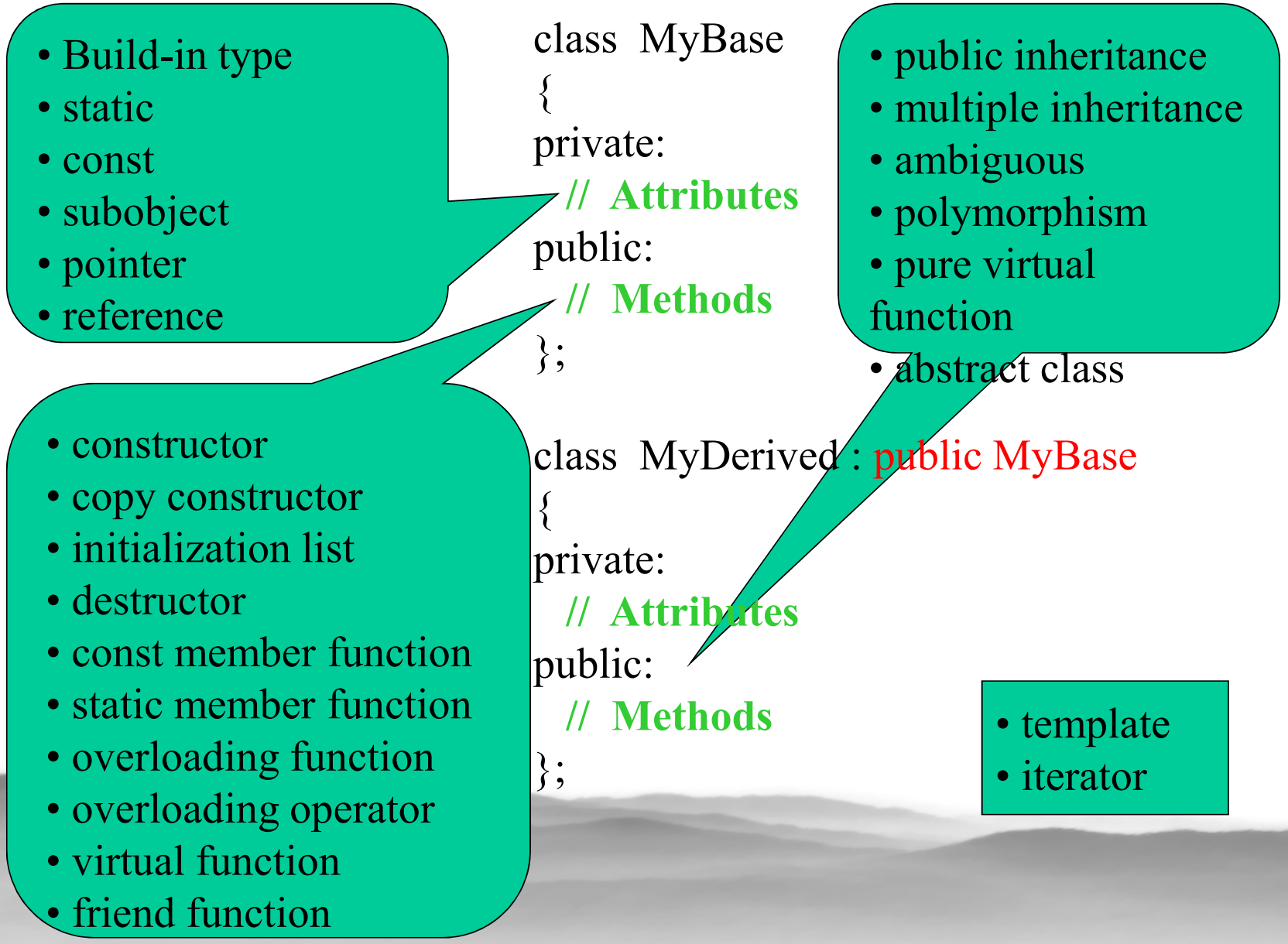
Beijing Institute of Technology

Review

Hu Sikang


• Build-in type
• static
• const
• subobject
• pointer
• reference
class MyBase
{
private:
// Attributes
public:
// Methods
};
• public inheritance
• multiple inheritance
• ambiguous
• polymorphism
• pure virtual function
• abstract class
• constructor
• copy constructor
• initialization list
• destructor
• const member function
• static member function
• overloading function
• overloading operator
• virtual function
• friend function
class MyDerived : public MyBase
{
private:
// Attributes
public:
// Methods
};
• template
• iterator

Part I

Types and Declarations

Types

- **Basis Types**

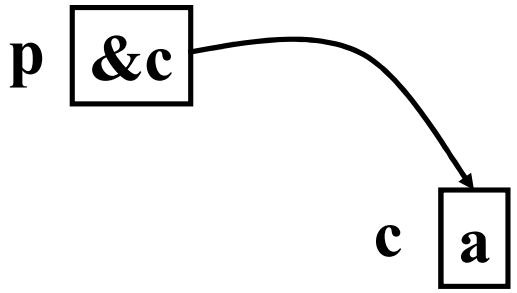
  char int float double void bool

The type of *void* isn't allowed to define variables. It's only used to denote the returning value of function.

Pointers

For a type *T*, *T is the type of "pointer to T". That is, a variable of type *T** can hold the address of an object of type *T*.**

```
char c = 'a';
//p holds the address of c
char* p = &c;
```



Pointers and Arrays

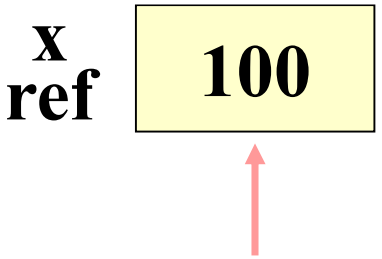
- Pointer is related with a type or a class.
- The pointer point at an array can be incremented, decremented.

```
#include <iostream.h>
void main()
{
        char ch[ ] = "I love China!";
        char *p = ch;
        while (*p)      cout << *p++;
        cout << endl;
}
```

Reference

A reference is an alternative name for an object.

```
//using reference in C++
#include <iostream.h>
void main()
{
   int x = 100;
   int &ref = x;
   cout << "x = " << x << endl;
   cout << " ref = " << ref << endl;
}
```



The variables, *x* and *ref,* express the same memory.

Dynamic Allocation

- "new" is used to dynamically allocate memory.
- "delete" is used to dynamically release memory.

```
#include <iostream.h>
void main()
{
        char *p;
        p = new int;
        *p = 10;
        cout << "Dynamically allocate memory." << endl;
        delete p;
}
```

What's the meaning?

p = new int[10];

p = new int(10);

Keywords: Const

When defining a constant, we use macro definition: #define in C. But in C++ we use a new keyword: *const*. *const* is often used when the value cannot be changed.

#define PI = 3.14 ⟶ const double PI = 3.14;

So programmer knows which type the PI is. With macro definition, PI is only a symbol which means 3.14 but not a double.

Static

- If you want a value to be extant throughout the life of a program, you can define a function's local variable to be **static** and **give it an initial value**.
- The initialization is performed only the first time the function is called, and the data retains its value between function calls.
- A **static** variable is unavailable outside the scope of the function.

10

Part Ⅱ

Functions

Argument Passing

When a function is called, store is set aside for its formal arguments and each formal arguments initialized by its corresponding actual argument. In C++, there are three main methods to transfer arguments:

1. ***Call By Value***
2. ***Call By Pointer***
3. ***Call By Reference***

Value Return

1. void pointers: to obtain flexibility of types

```
double f() { int a = 5;   return a; }

void* fp()
 {
     int local = 1;
     return &local;     // bad
 }
```

Value Return

2. Return a variable's reference

```
#include <iostream.h>
int val;
int& fun( ) {
      return val;
}
void main( )  {
      fun( ) = 100;
      cout << val << endl;
}
```

```
#include <iostream.h>
int val;
int& fun( )
{    int val = 10;
      return val;     //bad
}
void main( )  {
      cout << fun( ) << endl;
}
```

Overloaded Functions Names

Using the same name for operations on different data types or different arguments number is called *overloading*.

Especially, as a member function of a class, the keyword, *const*, judge the overloaded function.

Namespace

A *namespace* is a mechanism for expressing logical grouping. That is,if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact.

Namespace

```
#include <iostream.h>
namespace calculator
{
   double Add(double x, double y) {
         return x + y;
   }
   void Print(double x) {
         cout << "Result is " << x << endl;
   }
}
```

```
double x = calculator::Add(10, 20);

using namespace calculator;
double x = Add(10, 20)
```

Part III

Objects and Classes

Controlling Member Access

```
class class_name{
public:
//public members
protected:
//protected members
private:
//private members
};
```

the interface of this class

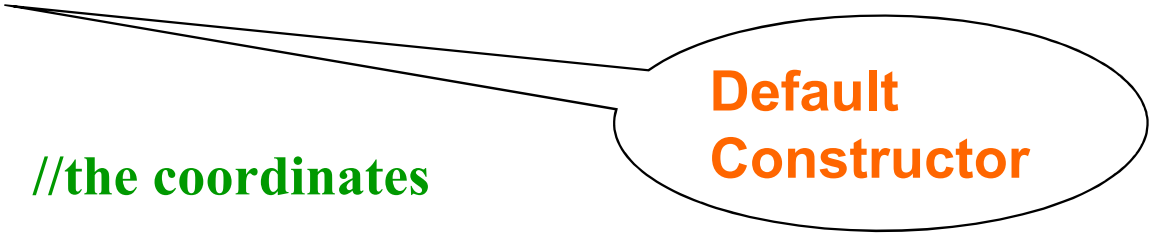
the interface of derived class

the inner attributes of this class

Constructors

- *Constructor* is recognized by having the same name as the class itself.
- *Constructor* is called by C++ automatically.
- *Constructor* is called to create an object.
- If you don't define constructor, C++ provides a default constructors: no parameters.

```
class Point {
public:
    Point() { };  //You can define or not
        …
private:
   int x,y;       //the coordinates
};
```



Copy Constructors

When is copy-constructor called?

（1）initialize object.

（2）When the argument is an object, copy- constructor is called.

（3）When the returning type of function is class type, copy-constructor is called.

Detructors

Notes:

1. **Destructor doesn’t have returning type.**
2. **Destructor is called when an object is destroyed.**
3. **Destructor doesn’t have argument.**
4. **There is only one destructors in a class.**
5. **Destructors are called in the reverse order of constructors called.**

Pointer: *this*

```
class Point {
public:
    SetPoint(int x, int y)
    {
        this->X = x;
        this->Y = y;
    }
private:
   int x, y;
};
```

```
void main()
{
      Point  P;
      P.SetPoint(1, 2);
}
```

The pointer, *this*, is assigned by P's address.

Static Members

- Declared with "static"
- Both member data and member functions may be declared static.
  - Only one copy exists to all objects of this class
  - Be controlled by the enclosing class
  - A static function can only access directly the static members .

Part IV

Operator Overloading

Operator Overloading

- **It refers to the technique of ascribing new meaning to standard operators such as +, -, = … when used with class operands.**
- ***In fact, it is a way to name a function.***
- ***Using the same name with some normal operators, make the programming more readable.***

Operator Overloading

```
class complex {
public:
    complex(double x = 0, double y = 0) {
        re = x;     im = y;
    }
    complex operator + (complex &c) {
        re += c.re;     im += c.im;    return *this;
    }
    complex operator - (complex &c) {
        re -= c.re;     im -= c.im;    return *this;
    }
private:
     double re, im;
};
```

```
void main()
{
    complex c, c1, c2;
            c = c1 + c2;
    c = c1 - c2;
}
```

c = c1 + c2

c = c1.operator+(c2);

Friend

A *friend function* is a function which can access the private member of a class. But it doesn't belong to a class. In fact, it's not a local function.

A *friend class* is a class in which all member functions have been granted full access to all the(private, protected, and certainly public) members of the class defining it as a friend(by instances).

Friend

Summary:

- **If overloading unary operators, we must define these as member functions.**
- **If overloading binary operators, we may define these as member functions or friend functions**
- **A *friend* declaration can be placed in either the private or the public part of a class declaration; it does not matter where.**

Conversion Operators

What is the conversion operators? For example:

```
complex obj(10, 20);
int x;
x = obj;   //conversion operators
```

So if we specify:
[1] an implicit conversion from a user-defined type to a basic type (because the basic types aren't classes), or
[2] a conversion from a new class to a previously defined class (without modifying the declaration for the old class), we can use conversion operators.

Conversion Operators

```
#include <iostream>
using namespace std;
class Rational {
public:
   Rational(double x = 0, double y = 1)
   {
      Numerator = x;
      Denominator = y;
   }
   operator double( ) {
         return Numerator / Denominator;
    }
private:
      double Numerator, Denominator;
};
```

```
void main( )
 {
     Rational r(100, 200);
     double d = r;
     cout << d << endl;
}
```

call function (conversion operator)

Part V

Derived Class

Accessing Control: public

class manager : public employee;

If a derived class, *manager*, has a public base class *employee*, then:

[1] the object of *manager* can access the member functions and member data of *employee's public*.

[2] the member functions of *manager* can access the member functions and member data of *employee's public and protected*.

[3] the member functions and the object of *manager* cannot access member functions and data of *employee's private*.

Constructors and Destructors during derived

[1] Constructors and destructors cannot be inherited.

[2] If a base class has constructors, then a constructor must be invoked in derived class.

[3] Default constructors can be invoked implicitly.

[4] However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called.

[5] Arguments for the base class' constructor are specified in the definition of a derived class' constructor.

[6] In this respect, the base class acts exactly like a member function of the derived class.

Process of constructors and destructors during derived

Class objects are constructed from the *bottom to up*:

[1] first the base, then the members, and then the derived class itself.

They are destroyed in the *opposite order*:

[2] first the derived class itself, then the members, and then the base.

[3] Members and bases are constructed in order of declaration in the class and destroyed in the reverse order.

Virtual Functions

Why do we use Virtual Function?

```
#include <iostream.h>
class A {
public:
        void f() { g(); }
        void g() { cout << "It's A::g()." << endl; }
};
class B: public A {
public:    void g() { cout << "It's B::g()." << endl; }
};
```

```
void main()  {
        A a;
        B b;
        a.f();      //call A::f()
        b.f();      //call A::f()
}
```

```
virtual void g()  {
        cout << "It's A::g()." << endl;
}
```

Virtual Functions

[1] There must be the same function definition when overloading the virtual function. It includes same returning type, same function name, same arguments number and same argument type.

[2] The virtual function must be a member function.

[3] The friend function cannot be defined as a virtual function.

[4] Destructor can be defined as a virtual function, but constructor cannot.

Multiple Inheritance

A class can have more than one direct base class, that is, more than one class specified after the : in the class declaration.

The use of more than one immediate base class is usually called *multiple inheritance*.

Multiple Inheritance

Problem 1: If there is a same name function, *fun(),* in the base class A and the base class B, and the object of derived class C calls fun(), then which fun() you want to call?

```
class A
{
public:
    void fun();
};

class B
{
public:
   void fun();
};
```

```
class C : public A, public B
{   };

void main()
{
    C obj;
    obj.fun();    //ambiguous
}
```

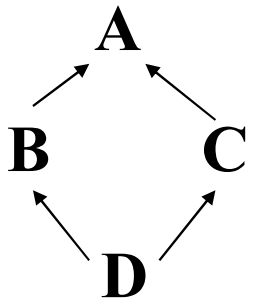
Multiple Inheritance

Problem 2: A derived class, *D*, has two base classes, and the two base classes have same base class A. When the object of class D calls the member function of class A, there will be a problem.

```
class A {
public:
        void fun();
};
class B : public A {
public:
        void FB();
};
class C : public A {
public:
        void FC();
};
```

```
class D : public B, public C
{ };

void main()
{
    D obj;
    obj.FB();    //ok
    obj.FC();    //ok
    obj.fun();   //ambiguous
}
```



Abstract Classes

A pure virtual function is a virtual function that contains a pure-specifier, designated by the "=0". It's used to be defined as a interface of derived class.

```
class  Number        //Abstract class
{
 public :
        Number ( int  i )  { val = i ; }
        virtual   void   Show ( ) = 0 ;   //pure function
protected :
        int   val ;
} ;
```

Part VI

Templates

Templates

- **Templates give us the means of defining a family of functions or classes that share the same functionality but which may differ with respect to the data type used internally.**
- **A class template is a framework for generating the source code for any number of related classes.**
- **A function template is a framework for generating related functions.**

Class Templates

One parameter:

- **Declare and define an object:**

```
template <class T>
class MyClass{
    T  val;
    //……
}
MyClass <int> x;
MyClass <student> aStudent;
```

Function Templates

- **A function can be defined in terms of an *unspecified type*.**
- **The compiler generates separate versions of the function based on the type of the parameters passed in the function calls.**

iterators

An *iterator* is an object that moves through a container of other objects and selects them one at a time, without providing direct access to the implementation of that container.

***Iterators* provide a standard way to access elements, whether or not a container provides a way to access the elements directly.**

Thanks for coming to my lessons!