



## 3.2 程序设计语言中的OOP机制

### 3.2.6 多态（方法名的过载）

- 前面已经讲过，如果需要对一个类的实例生成采用一种以上的处理语义，而 **Constructor** 的命名规则决定了不能采用不同的方法名，因此必须支持 **Constructor** 的过载多态。
- 如果能支持 **Constructor** 的过载多态，自然应支持其他方法名的过载多态。



## 3.2 程序设计语言中的OOP机制

### 3.2.6 多态（方法名的过载）

```
#include "Employee.h"
class Manager: public Employee {
    int    Level;
public:
    virtual void changeLevel( int l );
    void retire();
    Manager( char *n, int a, int l );
    Manager( char *n, int a,
              char *addr, char *tel,
              int l );
    Manager ();
    ~Manager ();
};
```

```
class Employee {
protected:
    char *Name;
    int   Age;
    char *Address;
    char *Telephone;
public:
    void change(int newAge);
    void change(char *newAddr);
    virtual void retire();
    Employee( char *n, int a );
    Employee( char *n, int a,
              char *addr, char *tel );
    Employee ();
    ~Employee ();
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.6 多态（方法名的过

```
#include "Employee.h"
class Manager: public Employee {
    int Level;
public:
    virtual void changeLevel( int l );
    void retire();
    Manager( char *n, int a, int l );
    Manager( char *n, int a,
             char *addr, char *tel,
             int l );
    Manager();
    ~Manager();
};
```

```
#include "Manager.h"
Manager::Manager( char *n, int a, int l )
    : Employee( n, a )
{ // ... }
Manager::Manager( char *n, int a,
                  char *addr, char *tel, int l )
    : Employee( n, a, addr, tel )
{ // ... }
Manager::Manager() : Employee()
{ // ... }
```

```
Employee( char *n, int a );
Employee( char *n, int a,
          char *addr, char *tel );
Employee();
~Employee();
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.6 多态（方法名的过载）

- 实现方法名过载的一种手段：
  - 将参数类型按既定规则依次转换成符号名，在方法名换名时加入，使得名称相同、参数数量或类型或顺序不同的方法，在换名后成为不同的方法名。

```
class Employee {  
    // ...  
public:  
    void change(int newAge);  
    void change(char *newAddr);  
    // ...  
};
```

→ void **\_Employee\_changeFI\_**(int)  
→ void **\_Employee\_changeFCP\_**(char\*)



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- 既然类已经能够限定方法的作用域，就应当允许对语言已经占用的操作符过载。

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三", 24);
  int i;
  // ...
  ++ e1;
  // ...
  for (i = 0; i < 10; i++)
  // ...
}
```

根据变量类型执行对应的操作

```
class Employee {
protected:
    char *Name;
    int   Age;
    char *Address;
    char *Telephone;
public:
    // ...
    Employee & operator ++()
    { Age ++; return *this; }
    // ...
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- C++允许在用户自定义类型中过载定义的操作符是：

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]





## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- C++不允许在用户自定义类型中过载定义的操作符是：

```
::      .      .*  
?:      sizeof typeid
```

- C++也不允许组合定义操作符，例如\*\*。



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- C++中可以采用三种方式来实现操作符的过载：成员函数方式；friend函数方式、普通函数方式。

成员函数方式

friend函数方式

```
class Employee {  
protected:  
    // ...  
public:  
    // ...  
    Employee & operator ++()  
    { Age ++; return *this; }  
    friend Employee & operator --  
    ( Employee & e)  
    { e.Age --; return e; }  
    // ...  
};  
bool operator== (Employee, Employee);
```

普通函数方式





## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- 在C++程序中调用过载定义操作符的方式，既可以是在表达式中引用操作符，也可以是函数调用：

```
class complex {  
    double re, im;  
public:  
    complex(double r, double i)  
        : re(r), im(i) { }  
    complex operator+ (complex c);  
    complex operator* (complex c);  
};  
  
bool operator== (complex, complex);
```

```
void f()  
{  
    complex a = complex(1, 3.1);  
    complex b(1.2, 2);  
    complex c = b;  
    a = b + c;  
    a = b.operator+(c);  
    bool e = a == b;  
    e = operator==(a, b);  
}
```



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- **friend 函数和 friend 类**：关于局部破坏封装的描述。
- 这种机制的合理使用，可以限定一个类具体到某个（些）其他类或函数的外部能见度（但是语言是无法防止滥用这种机制的）。

```
class a {  
    // 类 a 对于函数 f 和类 b 是不封装的。  
    friend void f(); friend class b;  
private:  
    int a1;  
    void g();  
    /* 类 a 没有public成员，意味着只有函数 f  
       和类 b 的实例能够访问 a 的实例 */  
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- An ordinary member function declaration specifies three logically distinct things :
  - The function can access the private part of the class declaration, and
  - the function is in the scope of the class, and
  - the function must be invoked on an object (has a **this** pointer).
- By declaring a member function **static**, we can give it the first two properties only.
- By declaring a function a **friend**, we can give it the first property only.



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- A binary operator can be defined by either a nonstatic member function taking **one argument** or a nonmember function taking **two arguments**.
  - 这意味着，定义为类C的成员函数的二元操作符，C的对象为第一操作数，该成员函数的形参为第二操作数。
- For any binary operator @, aa @ bb can be interpreted as either aa.operator@(bb) or operator@(aa, bb). 这取决于类型匹配情况。



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

```
class X {  
public:  
    void operator+ (int);  
    X(int);  
};  
void operator+ (X, X);  
void operator+ (X, double);  
void operator+ (X, int); ✗  
void f(X a, X b)  
{  
    a + 1;  
    1 + a; // 1 可以与X(1) 匹配  
    a + b;  
    a + 1.0;  
}
```

A constructor taking a single argument specifies a **conversion** from its argument type to the constructor's type.

A constructor requiring a single argument need not be called explicitly:

```
X xx = 3;  
// means: X xx = X(3);
```





## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- A unary operator, whether prefix or postfix, can be defined by either a nonstatic member function taking **no argument** or a nonmember function taking **one argument**.
  - 这意味着，定义为类C的成员函数的一元操作符，C的对象为操作数。
- For any prefix unary operator @, @aa can be interpreted as either aa.operator@( ) or operator@(aa). For any postfix unary operator @, aa@ can be interpreted as either aa.operator@(int) or operator@(aa, int)（这里的 **int** 是后缀一元操作符的标志参数）。





## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

```
class X {  
    // ...  
public:  
    X* operator& ();  
    X operator& (X);  
    X operator++ (int);  
    X operator& (X, X);✗  
    X operator/ ();✗  
};  
X operator- (X);  
X operator- (X, X);  
X operator-- (X&, int);  
X operator- ();✗  
X operator- (X, X, X);✗  
X operator% (X);✗
```

```
void f(X x)  
{  
    X* p = &x;  
    X* q = x.operator&();  
    X x2 = *p & x;  
    x2 = *p.operator&(x);  
    x2 = p->operator&(x);  
    X x3 = x++;  
    x3 = x.operator++(0);  
  
    x3 = -x;          // x3 = operator-(x);  
    x3 = x2 - x1;     // x3 = operator-(x2, x1);  
    x3 = x--;         // x3 = operator--(x, 0);  
}
```



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- C++的一些限制：
  - 过载定义操作符 =、[ ]、( )、->的必须是非静态成员函数，以保证其第一操作数一定是左值（lvalue）；
  - 不允许在过载定义操作符时使得所有操作数都是基本类型的，这也就是说，过载定义操作符时至少有一个操作数是自定义类型的。
  - 欲使第一操作数为基本类型者，不能定义为成员函数。



## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

- 采用成员函数和**friend**函数对操作符过载定义，其本质区别是前者是方法，可以被继承，而后者是一般函数，不可能被继承。





## 3.2 程序设计语言中的OOP机制

### 3.2.7 多态（操作符的过载）

```
class ary {  
    int idx;  
    int array[100];  
public:  
    ary & operator ++()  
        { idx++; return *this; }  
    int operator [] ( int i )  
        { return (i < 0 || i >= 100) ? 0 : array[i]; }  
    friend ary& operator +( ary &a1, ary &a2 );  
    ary() { idx = 0; }  
};
```

```
ary & operator +(ary &a1, ary &a2)  
{ static ary *ap = NULL;  
  int i;  
  if (ap == NULL)  
      ary *ap = new ary;  
  ap->idx = (a1.idx > a2.idx) ? a2.idx : a1.idx;  
  for (i = 0; i < ap->idx; i++)  
      ap->array[i] = a1.array[i] + a2.array[i];  
  return *ap;  
}
```

```
ary a1, a2, a3;  
int j;  
// ...  
++ a1;  
// ...  
j = a1[10];  
a3 = a1 + a2;
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 我们先来看一个例子：







## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

```
class StringVector {  
    String* v;  
    int sz;  
public:  
    String& operator[](int i);  
    int size() { return sz; }  
    StringVector(int vectorSize);  
    /* ... */  
};
```

```
class ComplexVector {  
    Complex* v;  
    int sz;  
public:  
    Complex& operator[](int i);  
    int size() { return sz; }  
    ComplexVector(int vectorSize);  
    /* ... */  
};
```

- 这两个类的内涵十分相似，都是向量，差别仅在于元素类型不同；所定义的数据结构和操作十分相似，但有差别的地方是类型而不是数据或地址。
- 如果能够把类型作为参数，就可能用另一个抽象结构产生这两个类乃至更多的向量类。





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 允许将类型作为参数的抽象结构称为类属。
- 支持类属的程序设计范型也被称为类属程序设计。
- 在C++中体现类属的机制是**模板**（*template*）。





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

```
class StringVector {  
    String* v;  
    int sz;  
public:  
    String& operator[](int i);  
    int size() { return sz; }  
    StringVector(int vectorSize);  
    /* ... */  
};
```

```
class ComplexVector {  
    Complex* v;  
    int sz;  
public:  
    Complex& operator[](int i);  
    int size() { return sz; }  
    ComplexVector(int vectorSize);  
    /* ... */  
};
```

```
template <class T> class Vector {  
    T* v;  
    int sz;  
public:  
    T& operator[](int i);  
    int size() { return sz; }  
    Vector(int vectorSize);  
    /* ... */  
};
```

Vector<String>

实例化

Vector<Complex>

实例化



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- The C++ template mechanism allows a type to be a parameter in the definition of a class (类模板) or a function (函数模板).
- 一个类是一组对象的抽象；一个类模板是一组类的抽象。
- 引入类模板的目的：对结构特征和行为特征相似、但成员的类型不能保证相同的一组类，进行高一级的抽象，以提高程序的重用性和规格化程度。
- 类模板的一个重要作用是对类库提供了强有力的支持。



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

指明C是一个类型参数（但可以不是类）

用类型参数来定义数据成员或成员函数

```
template<class C>
class String {
    struct Srep;
    Srep* rep;
public:
    String();
    String(const C*);
    String(const String&);
    C read(int i) const;
    // ...
};

template<class C>
struct String<C>::Srep {
    C* s;
    int sz;
    int n;
    // ...
};
```

```
// 对照:
class String {
    struct Srep;
    Srep* rep;
public:
    String();
    String(const char*);
    String(const String&);
    char read(int i) const;
    // ...
};

struct String::Srep {
    char* s;
    int sz;
    int n;
    // ...
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

指明C是一个类型参数（但可以不是类）

用类型参数来定义数据成员或成员函数

```
template<class C>
class String {
    struct Srep;
    Srep* rep;
public:
    String();
    String(const C*);
    String(const String&);
    C read(int i) const;
    // ...
};

template<class C>
struct String<C>::Srep {
    C* s;
    int sz;
    int n;
    // ...
};
```

// 类模板生成的实例（类）  
// 及其实例（对象）的生成  
// 与操作：

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;
```

```
class Jchar {
    // ...
};
String<Jchar> js;
```

// ...

```
char ch = cs.read(0);
unsigned char = us.read(1);
String<char> cs2 = "Hello";
// ...
```





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 一个模板的参数表中可以声明多个形参。
- 先声明的类型参数可以立即用来声明同一参数表中的其他形参。

```
template<class T, T def_val>  
class Cont {  
    // ...  
};
```





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 一个类模板可以继承另一个类模板的实例:

```
template<class T>
class Vec : public Vector<T> {
    // ...
};
```

- 一个类模板当然也可以继承另一个类:

```
template<class T>
class Teacher : public Person {
    // ...
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 一个类模板可以在继承另一个类模板的实例时，将自己实例化后得出的类型作为那个类模板的实参：

```
template<class T>
class Basic_ops {
    // ...
};

template<class T>
class Math_container : public Basic_ops<Math_container<T>> {
    // ...
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 一个类模板中的数据成员，可以用另一个类模板的实例来定义该数据成员的类型：

```
template<class T>
class Teacher : public Person {
    Vector<T*> group;
    // ...
};

Teacher<Student> t;
```

总之，所有允许出现类型的地方，都可以出现类模板的实例。



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 合法的模板实参:
  - a type;
  - a constant expression (a string literal is not acceptable);
  - the address of an object or function with external linkage:
    - `&of ;`      // *of* is the name of an object or a function
    - `f ;`      // *f* is the name of a function
  - a non-overloaded pointer to member:
    - `&X::of ;`    // *of* is the name of a member of class X.



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 合法的模板实参:
  - a type;
  - a constant expression (a string literal is not acceptable);
  - the address of an object or function with external linkage:
    - `&of ;`      *// of is the name of an object or a function*
    - `f ;`      *// f is the name of a function*
  - a non-overloaded pointer to member:
    - `&X::of ;`      *// of is the name of a member of class X.*

Why?





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 为什么对模板的实参有前面给出的限制？为什么用类模板产生类，与用这个类产生对象须同时进行？
- 关键在于用类模板产生类是在编译时而不是运行时进行。尽管类模板在表示和使用上很象一个类，但它们之间存在本质上的区别（至少对于C++语言是这样）：
  - 类模板的实例不是对象。
  - 一个类模板除了可以生成实例（类）以外，再没有任何操作可以作用于类模板的任何实例，其实例的状态（也就是一个类的基本结构）一经生成便不可改变。显然，无法用操作来改变状态的实体不是对象。





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- C/C++的多维数组是按行、连续存储的。

```
int a[3][5];  
int i, j, k;  
/* ... */  
k = a[i][j];
```

a[0][0]	
a[0][1]	
a[0][2]	
a[0][3]	
a[0][4]	
a[1][0]	
a[1][1]	
a[1][2]	
a[1][3]	
a[1][4]	
a[2][0]	
a[2][1]	
a[2][2]	
a[2][3]	
a[2][4]	

$a[i][j]$ 的地址计算公式： $i * \text{<列数>} + j$

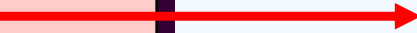


## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 任务：用类模板实现操作接口相同、按列存储的、任意元素类型的多维数组。

```
Array<int> a(3, 5);  
int      i, j, k;  
/* ... */  
k = a[i][j];
```



a[0][0]	
a[1][0]	
a[2][0]	
a[0][1]	
a[1][1]	
a[2][1]	
a[0][2]	
a[1][2]	
a[2][2]	
a[0][3]	
a[1][3]	
a[2][3]	
a[0][4]	
a[1][4]	
a[2][4]	

$a[i][j]$ 的地址计算公式： $i + j * \text{<行数>}$



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 错误的解（1）：

```
template<class T>
class Array {
    T* tpBody;
    int iRows, iColumns;
public:
    T& operator[] (int i, int j) { return tpBody[i + j * iRows]; }
    Array(int iRsz, int iCsz)
        { tpBody = new T[iRsz*iCsz]; iRows = iRsz; iColumns = iCsz; }
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 分析： [ ] 操作是二元操作，不能接受三个或更多个操作数。
- 因此必须分成两层类模板来定义，使得：

```
Array<int> a(3, 5);  
int      i, j, k;  
/* ... */  
k = a[i][j];
```

第1层类型

第2层类型



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 错误的解（2）：

```
template<class T>
class ArrayTmp {                // 第二层类型
    T* tpBody;
    int iRows, iColumns;
public:
    T& operator[](int j) { return tpBody[i + j * iRows]; }
    ArrayTmp (int iRsz, int iCsz)
        { tpBody = new T[iRsz*iCsz]; iRows = iRsz; iColumns = iCsz; }
};

template<class T>
class Array {                   // 第一层类型
    ArrayTmp<T> tTmp;           // 第一层类型与第二层类型的关联
public:
    ArrayTmp<T>& operator[](int i) { return tTmp; }
    Array(int iRsz, int iCsz) : tTmp(iRsz, iCsz) { }
};
```





## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 分析：需要解决下标值在不同类型的对象之间的传递问题：

```
Array<int> a(3, 5);  
int      i, j, k;  
/* ... */  
k = a[i][j];
```

第1层类型

下标i的传递

第2层类型



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 正确的解：

```
template <class T> class ArrayTmp { // 第二层类型
    friend class Array<T>;
    T*  tpBody;
    int iRows, iColumns, iCurrentRow;
    ArrayTmp(int iRsz, int iCsz)
        { tpBody = new T[iRsz*iCsz]; iRows = iRsz; iColumns = iCsz;
          iCurrentRow = -1; }
public:
    T& operator[](int j) { return tpBody[iCurrentRow + j * iRows]; }
};

template <class T> class Array { // 第一层类型
    ArrayTmp<T> tTmp;           // 第一层类型与第二层类型的关联
public:
    ArrayTmp<T>& operator[](int i) { tTmp.iCurrentRow = i; return tTmp; }
    Array(int iRsz, int iCsz) : tTmp(iRsz, iCsz) {}
};
```



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 正确的解：

```
template <class T> class ArrayTmp { // 第二层类型
    friend class Array<T>;          // 限定访问范围
    T*  tpBody;
    int iRows, iColumns, iCurrentRow;
    ArrayTmp(int iRsz, int iCsz)      // 从实例生成就设置了能见度控制
    { tpBody = new T[iRsz*iCsz]; iRows = iRsz; iColumns =
      iCurrentRow = -1; }
public:
    T& operator[](int j) { return tpBody[iCurrentRow + j * iRows]; }
};

template <class T> class Array {
    ArrayTmp<T> tTmp;
public:
    ArrayTmp<T>& operator[](int i) { tTmp.iCurrentRow = i; return tTmp; }
    Array(int iRsz, int iCsz) : tTmp(iRsz, iCsz) {}
};
```

计算元素位置

传递下标到下一层类型

二层类型

过渡到下一层类型



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类一例：按列存储的二维数组

- 从这个例子中得到的启示：
  - 不同层次（功能可能类似）的数据和操作，应当分离到不同的类/类模板中，再设计相应的关联结构和协同操作。
  - 用数据成员对象也可以表示类之间的层次关系。
  - 简化操作的接口有利于理解和使用，但需要基于相应的支撑结构和操作。反之，设计复杂接口的操作要容易得多，但可用性较差。



## 3.2 程序设计语言中的OOP机制

### 3.2.8 类属类

- 在类属类的设计中，要注意以下细节问题：
  - 在类属类中如果需要直接调用参数类型的操作，一定要确认它对所有的参数类型都是存在、并且接口完全相同的。
  - 对类属类的实例化的编译，一般是滞后到链接时才进行的。因此，如果类属类本身发生了修改，就需要编译那些与该类属类的实例化相关的所有文件，因此在 **make** 文件中要定义相应的依赖关系。







## 3.2 程序设计语言中的OOP机制

### 3.2.9 类属函数

- 使用模板时，由于类型参数不同而要求相应的操作不同。这是常见的，特别是比较操作和赋值操作。

```
template <class T>
void sort (Vector<T>& v) // 函数模板
{
    int n = v.size();
    for (int gap = n/2; 0 < gap; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i-gap; 0 <= j; j -= gap)
                if (v[j+gap] < v[j])
                {
                    T temp = v[j];
                    v[j] = v[i+gap];
                    v[j+gap] = temp;
                }
}
```

∴ 不同的类型 T 可能有不同的 < 和 = 操作



## 3.2 程序设计语言中的OOP机制

### 3.2.9 类属函数

- 因此，在定义模板时，如果其中需要使用与类型参数相关的操作，则需要：
  - 要求所有可能的实参类型都（过载）定义了这样的操作（但这通常是不可靠的），或者
  - 干脆在参数表中定义相应函数指针形参（需要统一规定函数的接口），在模板中调用其实参（与类型相关的操作的函数地址）以进行这样的操作。

可以借鉴 C 语言例程库中两个函数的做法：

```
void qsort(                const void *base, size_t nelem, size_t width,  
                int (*fcmp)(const void*, const void*));  
void * bsearch(const void *key, const void *base, size_t nelem, size_t width,  
                int (*fcmp)(const void*, const void*));
```



## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

- 例：完成满足下述要求的对象计数：
  - 要求对当前 Employee 对象和 Manager 对象的个数分别计数，其中对 Employee 对象计数时也应包括 Manager 对象。
- 分析：
  - 需要设置两个计数器，一个对所有的 Employee 对象和 Manager 对象计数，另一个只对 Manager 对象计数。
  - 计数应当与这两种对象的生成和消除同时进行。
  - 如何设置？



## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

- 思路1: 设立一个函数来专门控制 Employee 和 Manager 的实例生成, 以便在此过程中计数。
- 缺点: 非类型化; 不能保证对象的作用域; 难以实现对象生成时的初始化参数传入和多态等。

```
void EmpMngrInst (Employee **e,
                  Manager **m)
{
    static int Ecount = 0, Mcount = 0;
    if (*e == NULL)
    {
        *e = new Employee; ECount ++;
    }
    if (*m == NULL)
    {
        *m = new Manager; ECount ++;
        MCount ++;
    }
}
```





## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

- 思路2: 在类 Employee 和 Manager 中各增加一个数据成员作为计数器, 并在其 Constructor 和 Destructor 中分别定义有关它们的增值和减值操作。
- 这是错误的。
  - 画出这样的对象的存储结构就可以得出上述结论。





## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

- 这里提出了一种语言机制的支持要求：
  - 需要有允许一个类的所有实例共享的数据，而且这样的数据是以数据成员的方式出现的。
  - 这就是类属性。
- 这样的机制在 **Smalltalk-80** 中叫做类变量，在**C++**中叫做静态数据成员。



## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

```
#include "Employee.h"
class Manager: public Employee {
private:
    static int MCount;
    /* ... */
    Manager( char *n, int a, int l );
    Manager( char *n, int a,
             char *addr, char *tel,
             int l );
    Manager();
    ~Manager();
};
```

```
class Employee {
private:
    static int ECount;
    /* ... */
    Employee( char *n, int a );
    Employee( char *n, int a,
             char *addr, char *tel );
    Employee();
    ~Employee();
};
```

```
#include "Employee.h"
int Employee::ECount = 0;
Employee::Employee( char *n, int a )
{ ECount ++; /* ... */ }
/* ... */
Employee::~~Employee()
{ ECount --; /* ... */ }
```



## 3.2 程序设计语言中的OOP机制

### 3.2.10 数据成员值的共享

- 分析：
  - 对于对象计数器的修改只在 **Constructor** 和 **Destructor** 中定义，实现了封装的、与实例化过程同时进行的计数。
  - 满足了把 **Manager** 对象集合看成是 **Employee** 对象集合的一个子集这样的要求。
  - 仍具备类型化和抽象的特征，所有的 **Employee** 对象和 **Manager** 对象，不管其作用域如何，都具备这样的属性。
  - 用继承的方式来表示 **Employee** 对象集合与 **Manager** 对象集合之间的关系，对可扩充性也有很好的支持。



## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 问题:
  - 是否每个类都至少有一个实例?
- 回答:
  - 如果在类之间没有定义继承关系, 那么一个类至少应当有一个实例, 否则是冗余的类。
  - 如果存在继承关系, 则有可能需要在类层次结构的较高层次上存在着始终没有实例的类。





## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 设：在同一个应用中还需要引入学生这种实体。
- 显然，学生与雇员、经理等有一些共同的属性，如姓名、年龄、住址、电话等。





## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 思路1: 按照传统程序设计的做法, 可能会定义一个描述学生的类 Student, 在该类中再次定义上述属性及相关操作, 并定义其他与学生有关的属性和操作。
  - 在面向对象的应用中, 这种定义上的冗余显然是不合适的。
- 思路2: 把类 Student 定义成 Employee 的另一个子类, 以继承上述属性及相关操作。
  - 这虽然是行得通的, 但不应当提倡, 因为这里所蕴含的、与常识相悖的“学生 is\_a 雇员”的关系, 很难让其他人理解, 因而将影响对这个类的重用和维护。



## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 思路3: 引入一个新的类 Person, 把上述属性及相关操作从Employee 和 Student 中抽象出来, 定义在 Person 之中, 同时令 Employee 和 Student 继承 Person.
  - 这种做法弥补了思路 1 和思路 2 的不足, 是可取的。
  - 但是, 应用需求本身可能并没有对应于 Person 的实体, 这个类的出现是设计时追加的, 虽然反映了客观概念之间的关系, 但真实的目的是让子类来共享属性。
  - 因此, 这样的类只有内涵而没有外延, 就不应该允许它产生实例, 需要有机制加以抑制。



## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- C++中的相应机制：抽象类（*Abstract Classes*）
  - 抽象类：至少含有一个纯虚拟函数的类。
  - 纯虚拟函数：不用定义（也不允许定义）方法体的虚拟函数。







## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

抽象类

```
class Employee : public Person {  
    /* ... */  
    void PrintOn();  
    /* ... */  
};
```

```
class Student : public Person {  
    /* ... */  
    void PrintOn();  
    /* ... */  
};
```

```
class Person {  
protected:  
    char *Name;  
    int   Age;  
    char *Address;  
    char *Telephone;  
    int   PrintWidth;    // 输出宽度  
public:  
    virtual void PrintOn() = 0;  
    /* ... */  
};
```

纯虚拟函数





## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 抽象类对于软件结构设计的作用：
  - 规范了一组语义允许不同，语法必须或应当相同的操作接口。

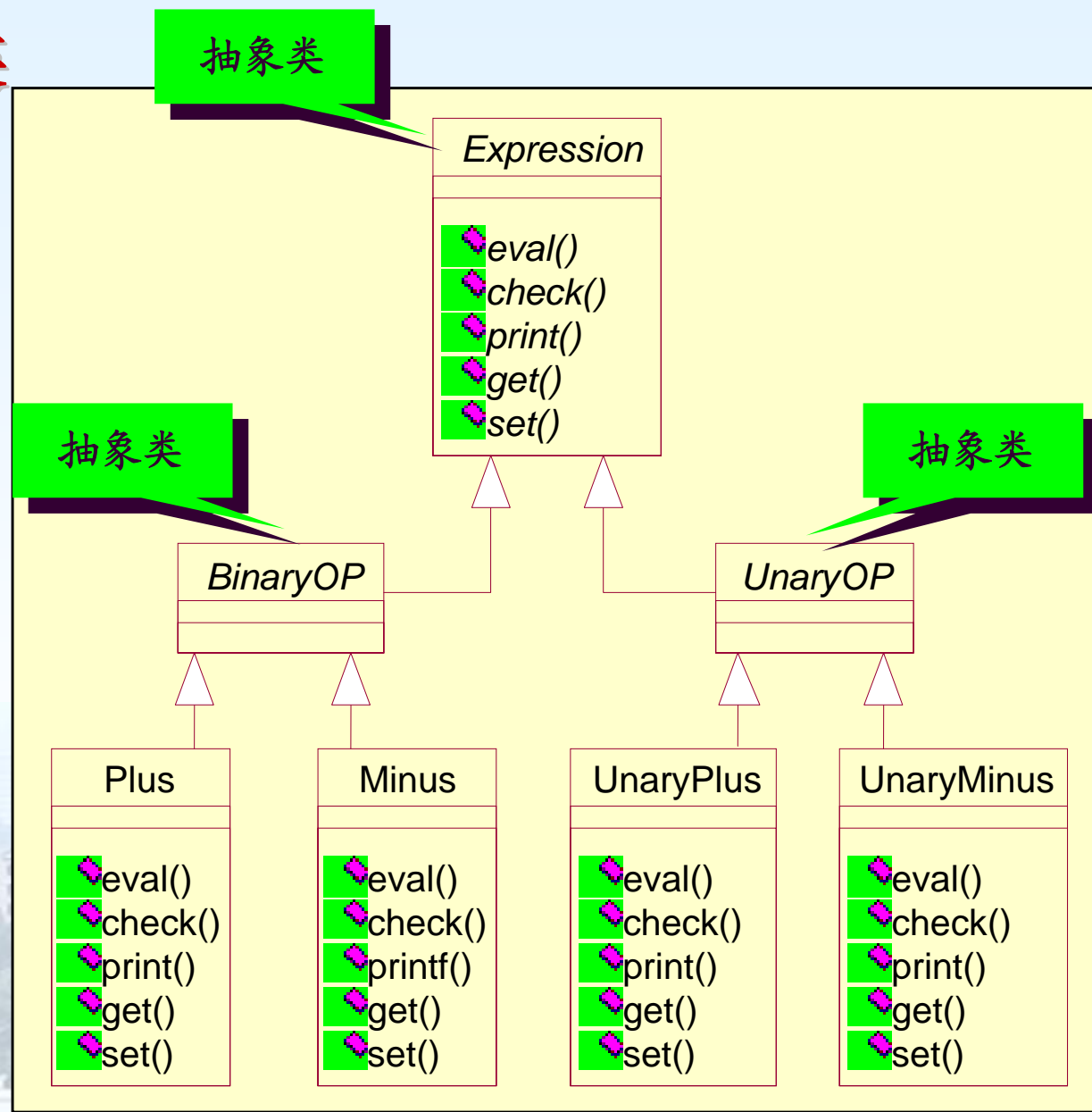




## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 例：设计和实现数学表达式的一种支撑机制。
- 每一种运算应提供的的能力：
  - 计算/求值；
  - 语法检查；
  - 以文本方式显示。
- 在不同的类中，同种运算的接口必须相同。

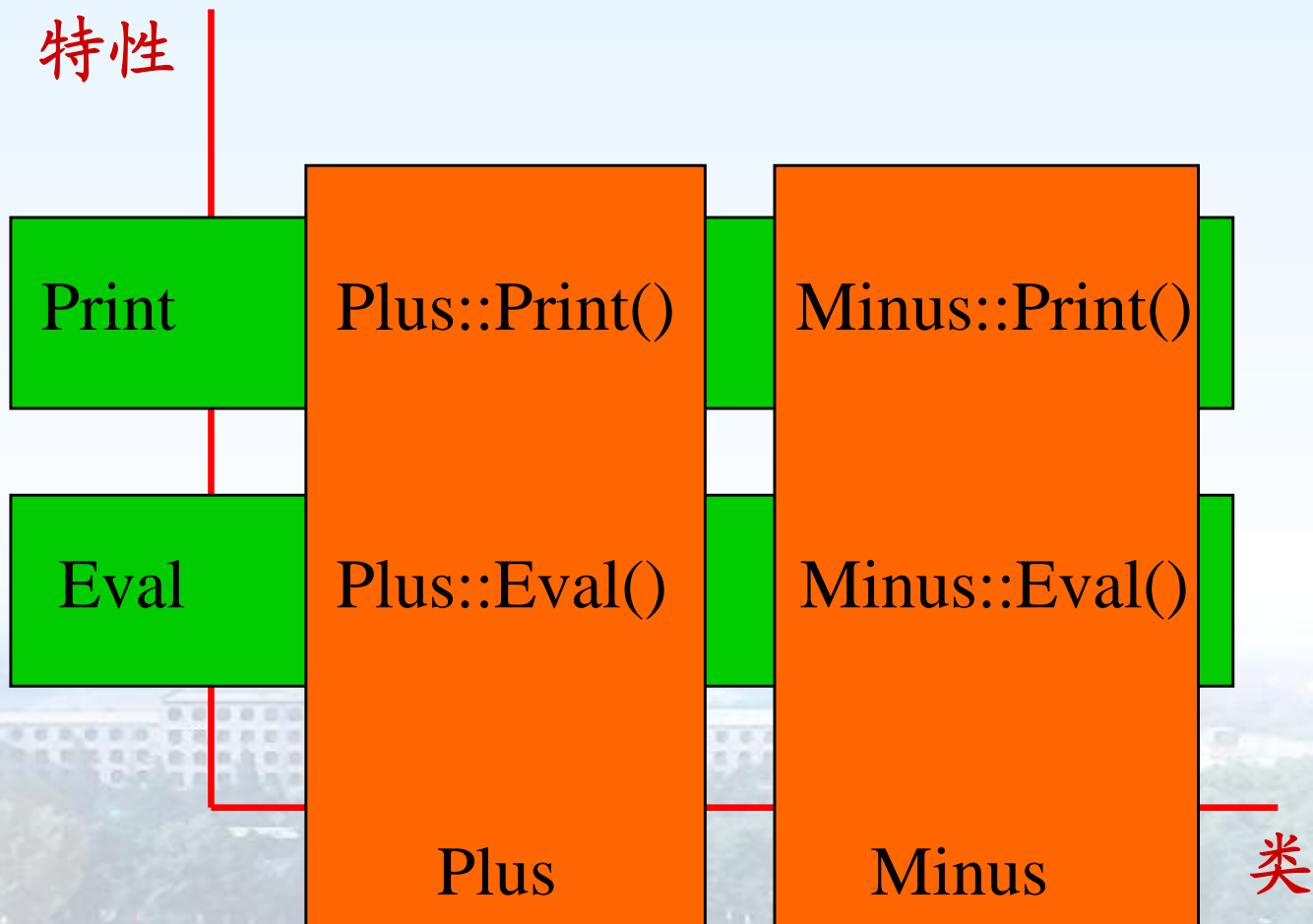




## 3.2 程序设计语言中的OOP机制

### 3.2.11 无实例的类

- 语法的统一有助于进行跨越类的特性规范化。





# 要点与引伸

- 在设计定义操作符重载的成员函数时，注意操作对象中谁是消息的接收者，谁是随消息传去的。
- 实际上，语言的基本类型的操作符，最终的动作都是用函数实现的，想象一下  $a + b$  等价于  $+(a, b)$ ，就不难理解操作符重载的实现方式。
- 类属类的实例是类，它定义的却不是作用于它的实例的操作，唯一允许作用于类属类的实例的操作，就是在编译执行的产生类的操作。



## 要点与引伸 (续)

- 静态数据成员和抽象类在程序设计时都具有双重作用：
  - 前者是使得静态数据既支持对象集合中对象的共享，又具备数据成员的特征；
  - 后者既定义了不允许有实例的类，又用纯虚函数的方式统一了那些子类肯定具备、父类又无法确定实现的操作的接口。
- 程序设计语言中有不少这类“一箭双雕”的聪明做法，值得我们效仿。





# 下一次课的内容

- 面向对象程序设计 - 程序设计语言中的**OOP**机制
  - 多重继承
- 面向对象程序设计 - **面向对象的程序**
  - 经典的程序实例：单链表
  - 定义有类型控制 and 操作投影的“外壳”
  - 利用统一接口的方法指针
- 面向对象程序设计 - **关于面向对象程序设计的讨论**
  - 什么是对象？
  - 什么是类？
  - 什么是继承性？
- 布置第二次作业



# 下一次课的内容 (续)

- 面向对象系统分析与设计 - 引言
  - 系统分析与设计的基本概念、重要性
  - 对系统分析人员的能力要求
- 面向对象系统分析与设计 - 信息系统及其基本特征
  - 什么是系统?
  - 信息系统
  - 信息系统的基本特征
  - 信息系统的分析与设计是十分棘手的工作