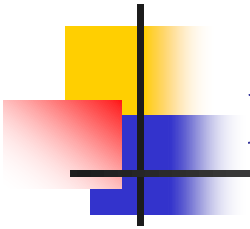
面向对象程序设计（part 3）



多态

- 同一论域中一个元素可有多种解释
- 提高语言灵活性

- 程序设计语言
  - 一名多用　　函数重载
  - 类属　　template

  - 00 程序设计　　虚函数

操作符重载

Compiler/Linker

- 函数重载
  - 名同，参数不同
  - 静态绑定

歧义控制

- 顺序
- 更好匹配 窄转换？

- 操作符重载
  - 动机

    操作符语义
    - built_in 类型 Compiler
    - 自定义数据类型 程序员
  - 作用
    - 提高可读性
    - 提高可扩充性

操作符重载

```
class Complex
{      double real, imag;
  public:
       Complex() { real = 0; imag = 0; }
       Complex(double r, double i) { real = r; imag = i; }
       Complex add(Complex& x);
};
Complex   a(1,2),b(3,4), c;
c = a.add(b);
```

```
class Complex
{    double real, imag;
     public:
      Complex() { real = 0; imag = 0; }
      Complex(double r, double i) { real = r; imag = i; }
      Complex operator + (Complex& x)
      {    Complex temp;
           temp.real = real+x.real;
           temp.imag = imag+x.imag;
           return temp;
      }
};
Complex   a(1,2),b(3,4),c;
c = a.operator +(b);
```

c = a + b

- 易理解
- 优先级
- 结合性

操作符重载

```
class Complex
{       double real, imag;
  public:
        Complex() { real = 0; imag = 0; }
        Complex(double r, double i) { real = r; imag = i; }
        friend Complex operator + (Complex& c1, Complex& c2);
};
Complex operator + (Complex& c1, Complex& c2)
{     Complex temp;
      temp.real = c1.real + c2.real;
      temp.imag = c1.imag + c2.imag;
      return temp;
}
Complex a(1,2),b(3,4),c;          c = a + b;
```

operator + (a, b)

至少包含一个用户自定义类型
(new、delete除外)

示例

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT};

Day& operator++(Day& d)
{  return d= (d==SAT)? SUN: Day(d+1); }

ostream& operator << (ostream& o, Day& d)
{           switch (d)
            {           case SUN: o << "SUN" << endl;break;
                        case MON: o << "MON" << endl;break;
                        case TUE: o << "TUE" << endl;break;
                        case WED: o << "WED" << endl;break;
                        case THU: o << "THU" << endl;break;
                        case FRI: o << "FRI" << endl;break;
                        case SAT: o << "SAT" << endl;break;
            }
            return o;
}
```

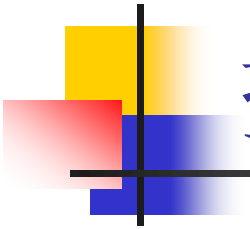
```
void main()
{       Day d=SAT;
        ++d;
       cout << d;
}
```

操作符重载

- 可重载的操作符
  - . .* :: ?:
- 基本原则
  - 方式
    - 类成员函数
    - 带有类参数的全局函数
  - 遵循原有语法
    - 单目/双目
    - 优先级
    - 结合性

```
class A
{    int x;
  public:
     A(int i):x(i){}
     void f() { … }
     void g() { … }
};
 void (A::*p_f)() ;

 p_f= &A::f;
(a.*p_f)();
```



操作符重载

- 双目操作符重载
  - 类成员函数
    - 格式

      <ret type> operator # (<arg>)
    - *this*　隐含
    - 使用

      <class name> a, b;

      a # b ;

      a.operator#(b) ;

操作符重载

- 全局函数
  - 友元

    friend <ret type> operator # (<arg1>,<arg2>)
  - 格式

    <ret type> operator # (<arg1>,<arg2>)
  - 限制

    = () [] → 不能作为全局函数重载

Why?

操作符重载

- 全局函数作为补充

obj + 10

10 + obj ?

```
class CL
{     int count;
  public:
        friend CL operator +(int i, CL& a);
       friend CL operator +(CL& a, int i);
};
```

操作符重载

- 永远不要重载 && 和 ||

```
char *p;
if ((p != 0) && (strlen(p) >10))  ...

if  (expressin1 && expression2) ...
if  (expression1.operator&&(expression2))
if  (operator &&(expression1, expression2)
```

操作符重载

```
class Rational {
    public:
        Rational(int,int);
        const Rational& operator *(const Rational& r) const;
    private:
        int n, d;
};
```

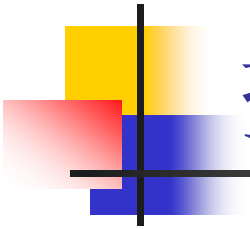
尽可能让事情有效率，
但不是过度有效率

operator *的函数体

- *return Rational(n*r.n, d*r.d);*
- *Rational *result = new Rational(n*r.n, d*r.d);*
  *return *result;*
- *static Rational result;*
  *result.n = n*r.n; result.d = d*r.d; return result;*

w = x*y*z

if ((a*b) ==(c*d))



操作符重载

- 单目操作符重载
  - 类成员函数
    - *this*　隐含
    - 格式

      <ret type> *operator # ()*

  - 全局函数

    <ret type> operator # (<arg>)

操作符重载

- *a++ vs ++a*
  - prefix *++* 左值

```
class Counter
{       int value;
    public:
        Counter() { value = 0; }
prefix operator  Counter& operator ++() // ++a
        {  value++;
           return *this;
        }                                  dummy argument
postfix operator Counter operator ++(int) //a++
       {    Counter temp=*this;
            value++;
            return temp;
        }
}
```

特殊操作符重载

- =
  - 默认赋值操作符重载函数
    - 逐个成员赋值(member-wise assignment)
    - 对含有对象成员的类，该定义是递归的
  - **赋值操作符重载不能继承** Why?

特殊操作符重载

```
class A
{      int x,y;
       char *p;
  public:
       A(int i,int j,char *s):x(i),y(j)
       { p = new char[strlen(s)+1]; strcpy(p,s);}
       virtual ~A() { delete[] p;}
       A& operator = (A& a)
       {  x = a.x; y = a.y;
          delete []p;
          p = new char[strlen(a.p)+1];
          strcpy(p,a.p);
          return *this;
       }
};
```

```
A  a, b;
a = b;
```

idle pointer

Memory leak

特殊操作符重载

- 避免自我赋值
  - Sample: class string
  - s = s?
    - class { … A void f(A& a); … }
    - void f(A&a1, A& a2);
    - int f2(Derived &rd, Base& rb);
  - Object identity
    - Content
    - Same memory location
    - Object identifier

```
class A
{   public:
        ObjectID  identity() const;
          …..
};
A *p1,*p2;
….
p1-> identity()== p2-> identity()
```

特殊操作符重载

- []

```
class string
{      char *p;
  public:
            string(char *p1)
            {    p = new char [strlen(p1)+1];  strcpy(p,p1); }
            char& operator [](int i) const { return p[i]; }
            const char operator [] (int i) const { return p[i]; }
            virtual ~string() { delete[] p;}
};
            …
string s("aacd");                s[2] = 'b' ;
const string cs("const");    cout << cs[0]; cs[0] = 'D';   ?
```

特殊操作符重载

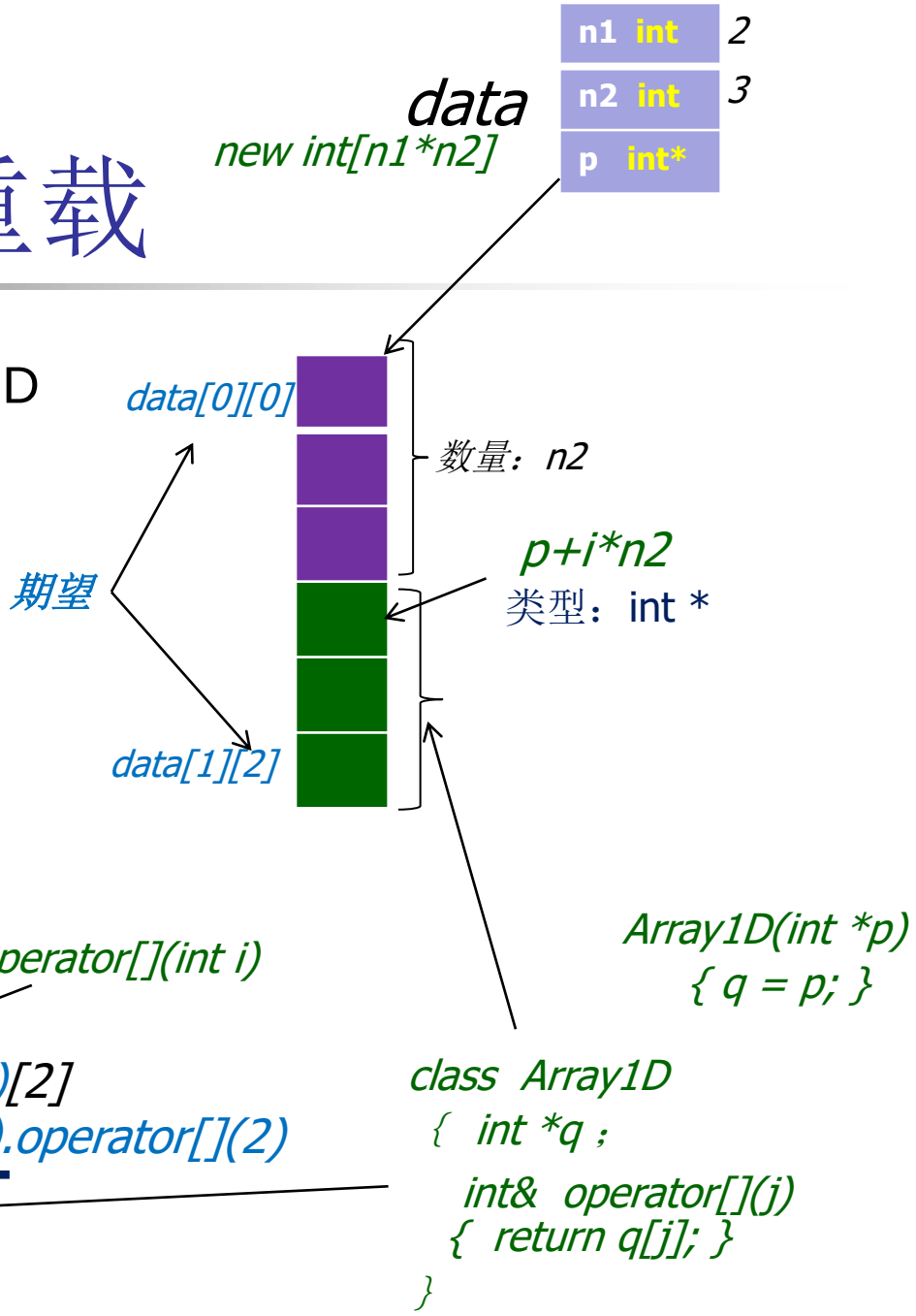
- 多维数组 class Array2D

```
class Array2D
{   int n1, n2;
    int *p;
public:
     Array2D(int l, int c):n1(l),n2(c)
    {  p = new int[n1*n2];  }
    virtual ~Array2D() { delete[] p; }
};
int & Array2D::getElem(int i, int j) { ... }

Array2D data(2,3);
data.getElem(1,2) = 0;
```

data[1][2] = 0;

?



proxy class
Surrogate
多维

```
class Array2D
{  public:
     class Array1D
     {    public:
            Array1D(int *p) { this->p = p; }
            int& operator[ ] (int index) { return p[index]; }
            const int operator[ ] (int index) const { return p[index]; }
          private:
            int *p;
     };
     Array2D(int n1, int n2) { p = new int[n1*n2]; num1 = n1; num2 = n2; }
    virtual ~Array2D( ) { delete [ ] p; }
     Array1D operator[ ] (int index) { return p+index*num2; }
     const Array1D operator[ ] (int index) const { return p+index*num2; }
 private:
     int *p;
     int num1, num2;
};
```

int *

特殊操作符重载

- ( )

```
class Func
{           double  para;
            int  lowerBound, upperBound;
  public:
            double operator () (double, int, int);
};
…
Func f;                    //函数对象
f(2.4, 0, 8);
```

```
class Array2D
{    int n1, n2;
     int *p;
public:
      Array2D(int l, int c):n1(l),n2(c)
     {  p = new int[n1*n2];  }
     virtual ~Array2D() { delete[] p; }
     int& operator()(int i, int j)
    {
         return  (p+i*n2)[j];
    }
};
```

特殊操作符重载

- 类型转换运算符
  - 基本数据类型
  - 自定义类

ostream f("abc.txt");
if (f) ....

重载 数值型：如 int

```
class Rational {
public :
        Rational(int n1, int n2) { n = n1; d = n2; }
        operator double() { return  (double)n/d; }
private:
        int  n, d;
};
```

减少混合计算中需要定义的操作符重载函数的数量

```
Rational  r(1,2);
double x = r;  x = x + r;
```

特殊操作符重载

- → *smart pointer*
  - →为二元运算符

重载时按一元操作符重载描述

```
A a;
a->f();
a.operator->( f) ??
a. operator ->()->f()
```

必须返回指针类型？

```
class CPen
{     int m_color;
      int m_width;
  public:
      void setColor(int c){ m_color = c;}
      int getWidth() { return m_width; }
};
class CPanel
{   CPen m_pen;
     int m_bkColor;
 public:
   CPen* getPen() {return &m_pen;}
   void setBkColor(int c) { m_bkColor =c; }
};
```

```
CPanel c;  c.getPen()->setColor(16);
c->setColor(16);
        //⇔ c.operator->()->setColor(16);
        //c.m_pen.setColor(16)
c->getWidth();
        //⇔ c.operator->()->getWidth();
        //c.m_pen.getWidth()

CPanel *p=&c;
p->setBkColor(10);
```

Prevent memory Leak

```
class A
{
  public:
        void f();
        int g(double);
        void h(char);
};
```

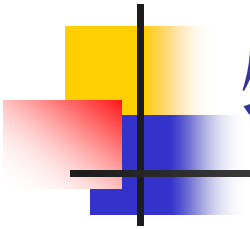
```
void test()
{
      AWrapper  p(new A);
      ……
      p->f();
      ……
      p->g(1.1);
      ……
      p->h('A');
      ……
      delete p;

}
```

局限性?

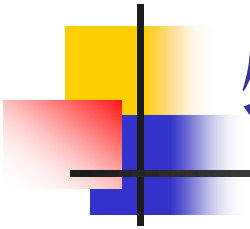
须符合**compiler**控制的生命周期

```
class AWrapper
{
   ? T   p;
  public:
       AWrapper(A *p) { this->p = p;}
       ~AWrapper() { delete p;}
       A*operator->() { return p;}
};
```



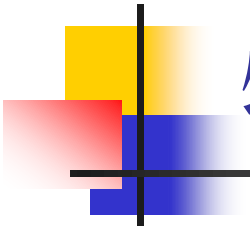
特殊操作符重载

- *new* 、*delete*
  - 频繁调用系统的存储管理，影响效率
  - 程序自身管理内存，提高效率
  - 方法
    - 调用系统存储分配，申请一块较大的内存
    - 针对该内存，自己管理存储分配、去配
    - 通过重载 *new* 与 *delete* 来实现
    - 重载的 *new* 和 *delete* 是静态成员
    - 重载的 *new* 和 *delete* 遵循类的访问控制，可继承



特殊操作符重载

- 重载 *new*
  - *void *operator new (size_t size, …)*
    - 名： *operator new*
    - 返回类型： *void **
    - 第一个参数：*size_t*（*unsigned int*）
      - 系统自动计算对象的大小，并传值给size
    - 其它参数：可有可无
      - *A *p = new (…) A* ， …表示传给*new*的其它实参
  - *new* 的重载可以有多个
  - 如果重载了*new*，那么通过*new* 动态创建该类的对象时将不再调用内置的（预定义的）*new*



特殊操作符重载

- 重载 *delete*
  - *void operator delete(void *p, size_t size)*
    - 名：*operator delete*
    - 返回类型：*void*
    - 第一个参数：*void **
      - 被撤销对象的地址
    - 第二个参数：可有可无；如果有，则必须是*size_t* 类型
      - 被撤消对象的大小
  - *delete* 的重载只能有一个
  - 如果重载了*delete*，那么通过*delete* 撤消对象时将不再调用内置的（预定义的）*delete*