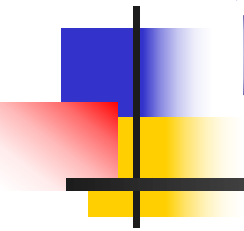
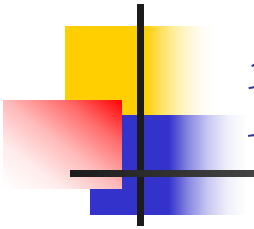


# 面向对象程序设计（part 3）



# 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

岐义控制

- 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;
}
```

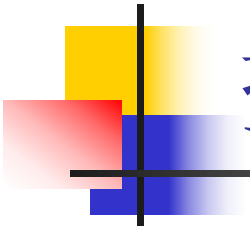
```
void main()
{       Day d=SAT;
        ++d;
       cout << d;
}
```

# 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;
};
```

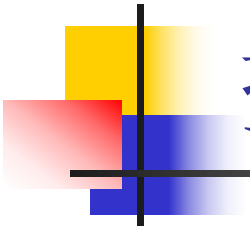
尽可能让事情有效率，
但不是过度有效率

**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';   ?
```

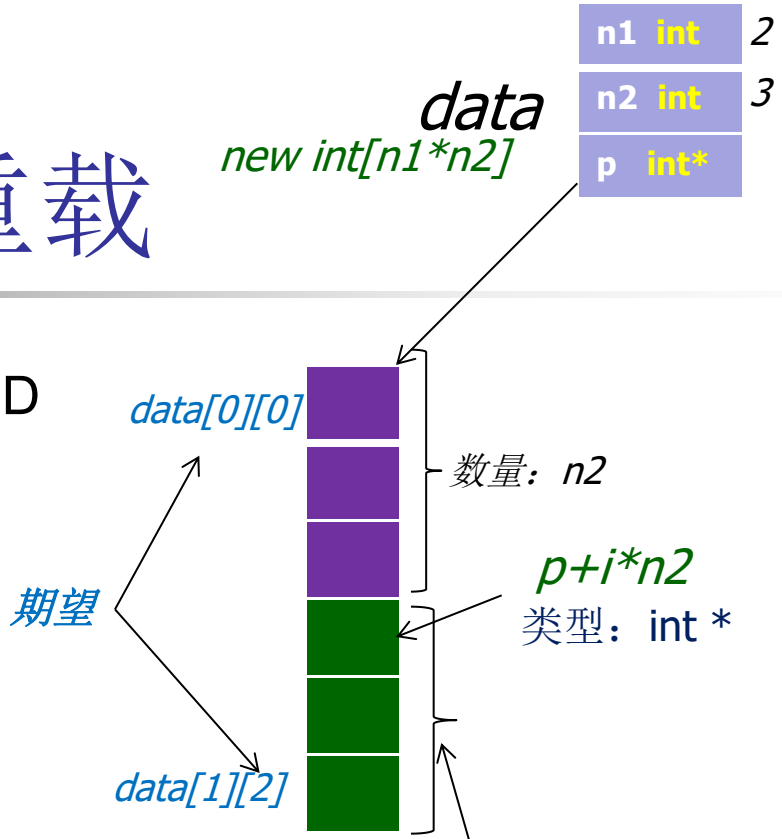
# 特殊操作符重载

- 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;**
**?**




int *operator[](int i)

data.operator[](1)[2]
data.operator[](1).operator[](2)

object

```
class  Array1D
 {  int *q ;
    int&  operator[](j)
   {  return q[j]; }
}
```

Array1D(int *p)
{ q = p; }

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

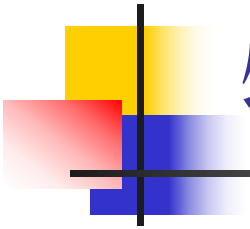
```
class AWrapper
{
   ? T   p;
  public:
       AWrapper(A *p) { this->p = p;}
       ~AWrapper() { delete p;}
       A*operator->() { return p;}
};
```

# 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

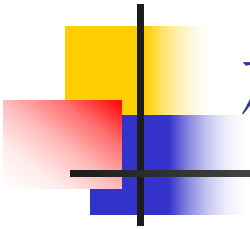
# 特殊操作符重载

- 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

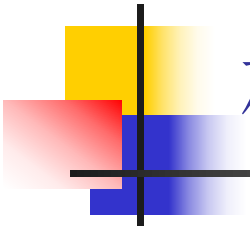
- 重载 *delete*
  - *void operator delete(void *p, size_t size)*
    - 名：*operator delete*
    - 返回类型：*void*
    - 第一个参数：*void **
      - 被撤销对象的地址
    - 第二个参数：可有可无；如果有，则必须是*size_t* 类型
      - 被撤消对象的大小
  - *delete* 的重载只能有一个
  - 如果重载了*delete*，那么通过*delete* 撤消对象时将不再调用内置的（预定义的）*delete*



# 模板 template

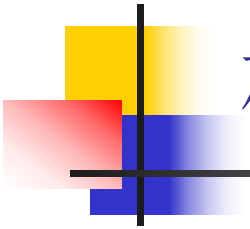
- 模板
  - 源代码复用机制
  - 参数化模块
    - 对程序模块（如：类、函数）加上**类型参数**
    - 对不同类型的数据实施相同的操作
  - 多态的一种形式

  - C++
    - 类属函数
    - 类属类



# 模板　template function

- 类属函数　　template function
  - 同一函数对不同类型的数据完成相同的操作
  - 宏实现
    - *#define  max(a,b)  ((a)>(b)?(a):(b))*
    - 缺陷
      - 只能实现简单的功能
      - 没有类型检查



# 模板

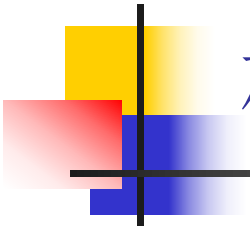
- 函数重载

  *int max(int,int);*

  *double max(double,double);*

  *A max(A,A);*

  - 缺陷
    - 需要定义的重载函数太多
    - 定义不全



# 模板

- 函数指针

*void sort(void * , unsigned int, unsigned int,*
*int (* cmp) (void *, void *) )*

  - 缺陷
    - 需要定义额外参数
    - 大量指针运算
    - 实现起来复杂
    - 可读性差

＝》template 引入的目标：
完全
清晰

# 模板

- 函数模板

```
template <typename T>
void sort( T  A[], unsigned int num)
{    for (int i=1; i<num; i++)
          for (int j=0; j<num-i; j++)
          {
              if  (A[j] > A[j+1])
              {      T  t = A[j];
                     A[j] = A[j+1];
                     A[j+1] = t;
               }
          }
}
```

```
int a[100];
sort(a,100);

double b[200];
sort(b,200);
```

```
class C { ... }
C a[300];
sort(a,300);
```

- 必须重载操作符 >
- =
- copy constructor

# 模板

- 函数模板定义了一类重载的函数
- 编译系统自动实例化函数模板
- 函数模板的参数
  - 可有多个类型参数，用逗号分隔
  - 可带普通参数
    - 必须列在类型参数之后
    - 调用时需显式实例化

```
template <class T1, class T2>
void   f(T1 a, T2 b）
{ …… }
```

```
template <class T, int size>
void f(T a)
{  T temp[size];   ……  }
……
f<int,10>(1);
```

# 模板

- 函数模板与函数重载配合使用

```
template <class T>
T max(T a, T b)
{ return a>b?a:b;
}

...
int  x, y, z;
double l, m, n;
z = max(x,y);
l = max(m,n);
```

定义一个max的重载函数：

```
double max(int a, double b)
{  return a>b? a : b; }
```

问题：*max(x,m)* 如何处理？

# 模板　template class

- 类属类

  类定义带有类型参数

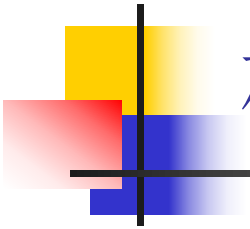
```
class Stack
{    int buffer[100];
  public:
     void push(int  x);
     int pop();
};
void Stack::push(int  x) { …  }

int  Stack::pop() { … }
……
Stack  st1;
```

```
template <class T>
class Stack
{    T buffer[100];
  public:
     void push(T x);
     T pop();
};
template <class T>
void Stack <T>::push(T x) { …  }

template <class T>
T Stack <T>::pop() { … }
……
Stack <int> st1;
Stack <double> st2;
```

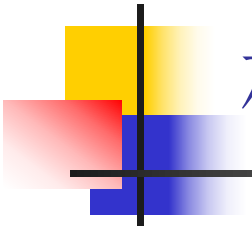
显式实例化



# 模板

- 定义了若干个类
- 显式实例化
- 可带有多个参数
  - 可带有普通参数
    - 逗号分隔
    - 须放在类型参数之后

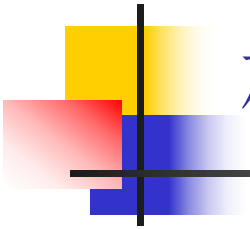
- 类模板中的静态成员属于实例化后的类



# 模板—例

```
template <class T, int size>
class Stack
{     T buffer[size];
  public:
       void push(T x);
       T pop();
};
template <class T, int size>
void Stack <T, size>::push(T x) { …  }

template <class T, int size>
T Stack <T, size>::pop() { … }
        ……
Stack <int, 100>         st1;
Stack <double, 200>   st2;
```



# 模板

- 模板是一种源代码复用机制
  - 实例化：生成具体的函数/类
  - 函数模板的实例化
    - 隐式实现
    - 根据具体模板函数调用
  - 类模板的实例化
    - 创建对象时显式指定

- 是否实例化模板的某个实例由使用点来决定；如果未使用到一个模板的某个实例，则编译系统不会生成相应实例的代码

# 模板

## C++中模板的完整定义通常出现在头文件中

- 如果在模块A中要使用模块B中定义的某模板的实例，而在模块B中未使用这个实例，则模块A无法使用这个实例

```
#include "file1.h"
template <class T>
void S<T>::f() { …}

template <class T>
T max(T x, T y)
{ return x>y?x:y;}

void main()
{  int a,b;
   max(a,b);
   S<int> x;
   x.f();
}
```

file1.cpp

```
template <class T>
class S
{    T a;
  public:
     void f();
};
```

file1.h

```
#include "file1.h"
extern double max(double, double);
void sub()
{  max(1.1, 2.2);          //Error
   S<float>  x;
   x.f();                  //Error
}
```

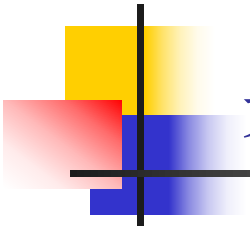
file2.cpp

# Template MetaProgramming

```
template<int  N>
class Fib
{ public:
   enum { value = Fib<N - 1>::value + Fib<N - 2>::value };
};

template<>
class Fib<0>
{ public:
   enum { value = 1 };
};
template<>
 class Fib<1>
{ public:
   enum { value = 1 };
};

void main()
{   cout << Fib<8>::value << endl;   // calculated at compile time
}
```



# 异常处理

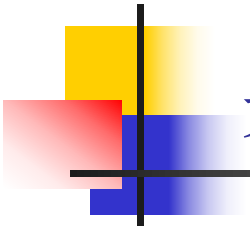
- 错误
  - 语法错误
    - 编译系统
  - 逻辑错误
    - 测试
- 异常　　*Exception*
  - 运行环境造成
    - 内存不足、文件操作失败等
  - 异常处理

# 异常处理

- 特征
  - 可以预见
  - 无法避免
- 作用
  - 提高程序鲁棒性（*Robustness*）

```
void f(char *str)
{    ifstream  file(str);
     if  (file.fail())
    { … //异常处理  }
    int x;
    file >> x;
      …
}
```

思考：发现异常之处与处理异常之处不一致，怎么处理？



# 异常处理

- 常见处理方法
  - 函数参数
    - 返回值
    - 引用参数
  - 逐层返回
- 缺陷
  - 程序结构不清楚

*构造函数执行出现异常?*

# 异常处理

- C++异常处理机制
  - 一种专门、清晰描述异常处理过程的机制

- 处理机制
  - *try*
    - 监控
  - *throw*
    - 抛掷异常对象
  - *catch*
    - 捕获并处理

```
try
{ <语句序列> }
```

```
throw <表达式>
```

```
catch (<类型> [<变量>])
{ <语句序列> }
```

# 异常处理

- *catch*
  - 类型：异常类型，匹配规则同函数重载
  - 变量：存储异常对象，可省

- 一个*try* 语句块的后面可以跟多个*catch* 语句块，用于捕获不同类型的异常进行处理

```
void  f()
{  ……
     throw 1;
   ……
    throw 1.0;
   ……
     throw "abcd";
   …….
}
```

```
try
{    f();   }
catch ( int )          //处理throw 1;
{ … }
catch ( double )     //throw 1.0
{ … }
catch (char * )     //throw "abcd"
{ … }
```

# 异常处理

- 异常处理的嵌套

多层传播

$f \rightarrow g \rightarrow h$ 调用关系

```
f()
{    try
     {    g();    }
     catch (int)
     { … }
    catch (char *)
     { … }
}
```

```
g()
{    try
     {    h();  }
     catch (int)
     { …  }
}
```

```
h()
{    …
  throw 1;   //由g捕获并处理
     …
  throw "abcd"; //由  f捕获并处理
}
```

如所抛掷的异常对象在调用链上未被捕获，则由系统的*abort*处理

# 异常处理

- 定义异常类
  - 注意*catch* 块排列顺序

*Catch exceptions by reference*

```
class FileErrors{  };
class NonExist: public FileErrors {  };
class WrongFormat: public FileErrors{  };
class DiskSeekError: public FileErrors{  };
```

```
int  f()
{    try
     {   WrongFormat wf; throw wf; }
     catch (FileErrors&){{.....  }
     catch (NonExist&) { }. ... }
     catch (DiskSeekError&) { } ... }
}
```

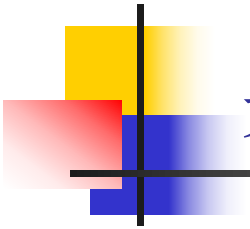
尝试多继承

```
class MyExceptionBase {
};

class MyExceptionDerived: public MyExceptionBase {
};

void f(MyExceptionBase& e) {
        throw e;
}

int main() {
        MyExceptionDerived e;
        try {
                f(e);
        } catch (MyExceptionDerived& e) {
                cout << "MyExceptionDerived" << endl;
        } catch (MyExceptionBase& e) {
                cout << "MyExceptionBase" << endl;
        }
}
```



# 异常处理

- 特例
  - 无参数*throw*
    - 将捕获到的异常对象重新抛掷出去

      *catch (int)  {  throw;  }*
  - *catch(…)*
    - 默认异常处理

*如何应对多出口引发的处理碎片？*

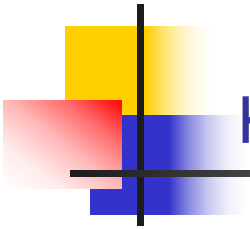
*对程序验证特征的支持*

```
template<class T, class E>
inline void Assert(T exp, E e)
{
    if (DEBUG)
        if (!exp) throw e;
}
```

*实现*

**不影响对象布局**

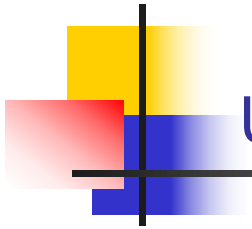
**程序状态 <-> 析构函数、异常处理器**



# Know what functions C++ silently writes and calls

- class Empty { };
- class Empty {

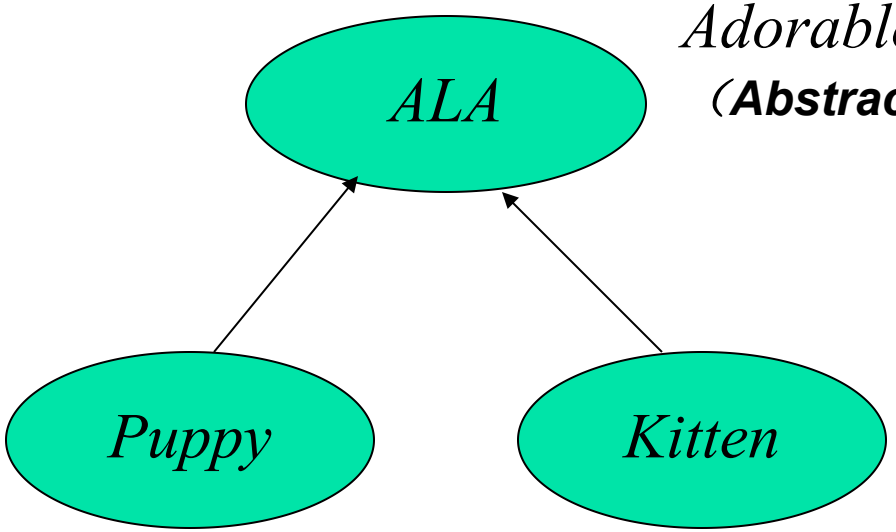
```
Empty();
Empty(const Empty&);
~Empty();
Empty& operator=(const Empty&);
Empty *operator &();
const Empty* operator &() const;
};
```



# Use destructors to prevent resource leaks

- Question -----resource leaks
  - 【小动物收养保护中心】
    - 收养中心每天产生一个文件，包含当天的收养个案信息
    - 读取这个文件，为每个个案做适当的处理

# Use destructors to prevent resource leaks




*Adorable Little Animal*

***(Abstract Base Class)***

```
class ALA
{ public:
      virtual void processAdoption() = 0;
                        ……..
};
```

```
class Puppy: public ALA
{ public:
      virtual void processAdoption();
                        ……..
};
```

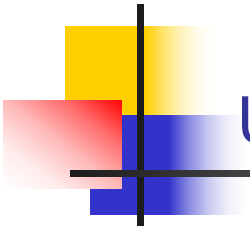
```
class Kitten: public ALA
{ public:
      virtual void processAdoption();
                        ……..
};
```

# Use destructors to prevent resource leaks

```
void processAdoptions(istream& dataSource)
{  while (dataSource)
   {   ALA *pa = readALA(dataSource);
       try
       {   pa->processAdoption();}
       catch (…)
       {   delete pa;   throw; }
       delete pa;
   }
}
```

- *结构破碎*
- *被迫重复“清理码”*

*集中处理？*



# Use destructors to prevent resource leaks

- Solution
  - Smart pointers
    - Template <class T>

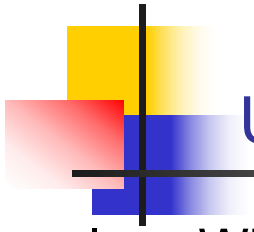
```
class auto_ptr
{  public:
        auto_ptr(T *p=0):ptr(p) {}
       ~auto_ptr() { delete ptr; }
        T* operator->()  const { return ptr;}
        T& operator *()  const { return *ptr; }
    private:
        T*  ptr;
};
```

# Use destructors to prevent resource leaks

```
void processAdoptions(istream& dataSource)
{  while (dataSource)
   {   auto_ptr<ALA>   pa(readALA(dataSource));
       pa->processAdoption();
   }
}
```

- 【GUI应用软件中的某个显示信息的函数】
  - void displayInfo(const Information& info)

```
{    WINDOW_HANDLE  w(createWindow());
     display info in window corresponding to w;
     destroyWindows(w);
}
```



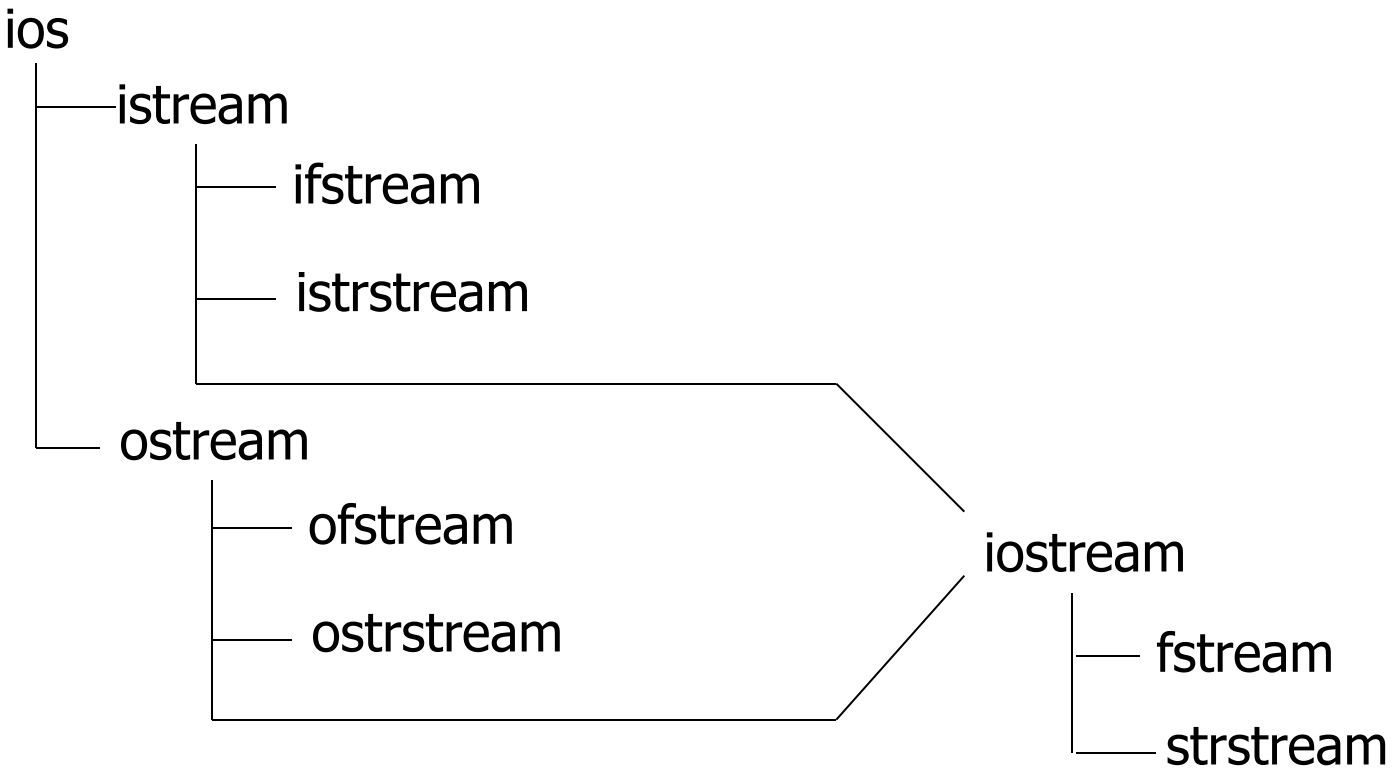
# Use destructors to prevent resource leaks

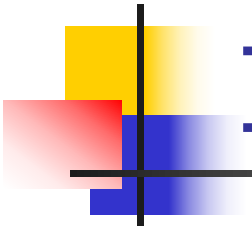
```
class WindowHandle
{ public:
    WindowHandle(WINDOW_HANDLE handler) : w(handler) {}
    ~WindowHandle() { destroyWindow(w); }
   operator WINDOW_HANDLE() { return w; }
  private:
     WINDOW_HANDLE  w;
    WindowHandle(const WindowHandle&);
    WindowHandle & operator = (const WindowHandle&);
};
```

```
void displayInfo(const Information& info)
 {    WindowHandle  w(createWindow())
      display info in window corresponding to w;
}
```

# I/O 处理

- 基于函数库的I/O
- 基于类库的I/O

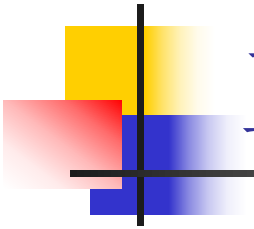




# I/O 处理

- I/O流库的三类输入/输出操作
  - 控制台I/O

    标准I/O设备
    - *cin、cout、cerr、clog*
  - 文件I/O
  - 字符串I/O



# 重定向

- *ifstream in("in.txt");*

  *streambuf *cinbuf = cin.rdbuf(); //save old buf*

  *cin.rdbuf(in.rdbuf()); //redirect cin to in.txt!*

  *ofstream out("out.txt");*

  *streambuf *coutbuf = cout.rdbuf(); //save old buf*

  *cout.rdbuf(out.rdbuf()); //redirect cout to out.txt!*

  *string word; cin >> word; //input from the file in.txt*

  *cout << word << " "; //output to the file out.txt*

  *cin.rdbuf(cinbuf); //reset to standard input again*

  *cout.rdbuf(coutbuf); //reset to standard output again*

  *cin >> word; //input from the standard input*

  *cout << word; //output to the standard input*

# I/O 处理

*Virtualizing non-member functions*

- 操作符<<和>>重载
  - 对自定义类的对象的I/O
  - 全局(友元)函数重载

```
class   CPoint2D
{    double   x,  y;
 public:
    friend ostream& operator << (ostream&, CPoint2D &);
};
ostream& operator << (ostream& out,  CPoint2D& a)
{           out << a.x << ",” << a.y << endl;
            return out;
}
...         CPoint2D   a;          cout << a;
```

```
class CPoint3D:  public CPoint2D
 {   double z;
     ...
friend   ostream&   operator << (ostream &, CPoint3D &);
 };
 ......
  CPoint3D  b;    cout  <<  b;
```

只显示*b.x*和*b.y*，而没显示*b.z*

```
ostream&   operator << (ostream& out,  CPoint3D & b)
{  out << b.x << ",” << b.y <<",”  << b.z << endl;
                    return out;    }
```

# I/O 处理

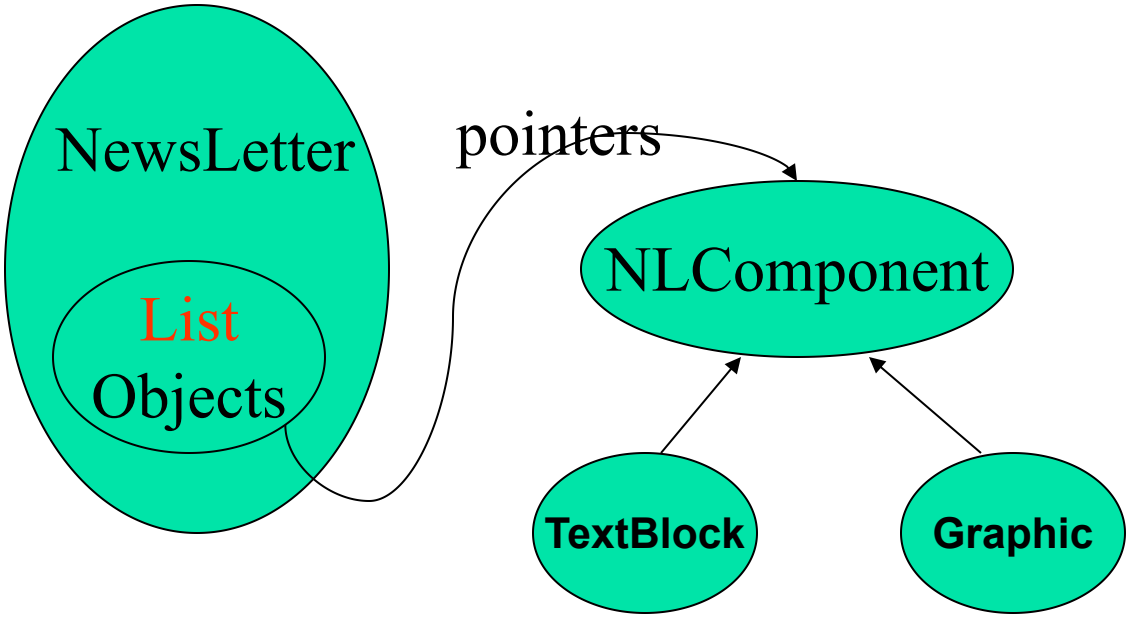
```
class CPoint2D
{       double  x,  y;
     public:
          …
        virtual void display(ostream& out)
        {   out << x << ',' << y << endl;  }
};

ostream& operator << (ostream& out,  CPoint2D &a)
{    a.display(out);   return out;  }

class CPoint3D: public CPoint2D
{       double  z;
    public:
        …
         void display(ostream& out)
        {    CPoint2D::display();   out << ','<< z << endl;  }
};
```

# Virtualizing constructors

- Virtual constructor
  - Virtual function
  - Constructor
- Question
  - 【报纸】
    - 文字、图形

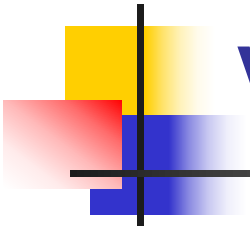


# Virtualizing constructors

```
class NLComponent {…};
class TextBlock : public NLComponent {…};
class Graphic :    public NLComponent {…};
class NewsLetter
{   public:
        NewsLetter(istream& str)
        {   while (str)   components.push_back(readComponent(str));  }
        static NLComponent * readComponent(istream& str);
    private:
        list<NLComponent *> components;
}
```

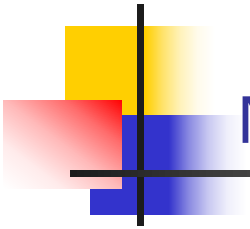
```
NewsLetter(const NewsLetter& rhs)
{  for (list<NLComponent *>::iterator it=rhs.component.begin();
                                    it != rhs.component.end(); ++it )
        component.push_back(  ??? );
}
```

new  TextBlock? Graphic?



# Virtualizing constructors

- virtual NLComponent * clone() const = 0;
- virtual TextBlock * clone() const
  { return new TextBlock(*this); }
- virtual Graphic * clone() const
  { return new Graphic (*this); }
- NewsLetter::NewsLetter( const NewsLetter& rhs)
  { for ( list<NLComponent *>::iterator it=rhs.component.begin();
  it != rhs.component.end(); ++it )
  component.push_back((*it)->clone());
  }



# Never treat arrays polymorphically

- Question

```
class BST { … };
class BalancedBST: public BST { … };

void printBSTArray(ostream& s, const BST array[],
   int numElements)
{  for (int i=0; i < numElements; i++)  s << array[i]; }

BalancedBST bBSTArray[10];
…
printBSTArray(cout, bBSTArray, 10);
```