程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2024 年 4 月 10 日

大纲

类型强制转换

- static_cast
- interpret_cast
- const_cast
- dynamic_cast

static_cast

static_cast 作用和 C 语言风格强制转换的效果基本一样. 主要用法：

- 用于类层次结构中基类和派生类之间指针或引用的转换
  进行上行转换（把派生类的指针或引用转换成基类表示）是安全的
  进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的
- 用于基本数据类型之间的转换，如把 int 转换成 char。这种转换的安全也要开发人员来保证

如果涉及到类的话，static_cast 只能在有相互联系的类型中进行相互转换.
static_cast 不能用于整型和指针之间的互相转换。

static_cast

```cpp
#include <iostream>
using namespace std;
class A {
public:
  operator int() { return 1; }
  operator char * (){ return NULL; }
};
int main(){
  A a;
  int n; char * p = "New Dragon Inn";
  n = static_cast<int>(3.14);  // n 的值变为 3
  n = static_cast<int>(a); //调用 a.operator int, n 的值变为 1

  p = static_cast<char*>(a);
  //调用 a.operator char *,p 的值变为 NULL
  n = static_cast<int> (p);
  //编译错误，static_cast 不能将指针转换成整型
  p = static_cast<char*>(n);
  //编译错误，static_cast 不能将整型转换成指针
  return 0;
}
```

reinterpret_cast

reinterpret_cast 用来进行各种不同类型的指针之间的转换、不同类型的引用之间转换、以及指针和能容纳得下指针的整数类型之间的转换。转换的时候，执行的是逐个比特拷贝的操作。

reinterpret_cast

```cpp
#include <iostream>
using namespace std;
class A {
public:
  int i;
  int j;
  A(int n):i(n),j(n) { }
};
int main() {
  A a(100);
  int & r = reinterpret_cast<int&>(a); //强行让 r 引用 a
  r = 200;  //把 a.i 变成了 200
  cout << a.i << "," << a.j << endl;   // 输出 200,100
  int n = 300;
  A * pa = reinterpret_cast<A*> ( & n); //强行让 pa 指向 n
  pa->i = 400;  // n 变成 400
  pa->j = 500;  //此条语句不安全，很可能导致程序崩溃
  cout << n << endl;  // 输出 400
  long long la = 0x12345678abcdLL;
  pa = reinterpret_cast<A*>(la);
  long long u = reinterpret_cast<long long>(pa); //pa 逐个比特拷贝到 u
  cout << hex << u << endl;
  typedef void (* PF1) (int);
  typedef int (* PF2) (int,char *);
  PF1 pf1;
  PF2 pf2;
  pf2 = reinterpret_cast<PF2>(pf1); //两个不同类型的函数指针之间可以互相转换
  return 0;
}
```

const_cast

用来进行去除 const 属性的转换。将 const 引用转换成同类型的非 const 引用，将 const 指针转换为同类型的非 const 指针时用它。例如：

```
1  int main() {
2    const int a = 10;
3    const int * p = &a;
4    int *q = const_cast<int *>(p);
5    *q = 20;     // “*q=20”语句为未定义行为
6    cout <<a<<" "<<*p<<" "<<*q<<endl;//a=10 *p=20 *q=20
7    return 0;
8  }
```

dynamic_cast

dynamic_cast 专门用于将多态基类的指针或引用，强制转换为派生类的指针或引用，而且能够检查转换的安全性。对于不安全的指针转换，转换结果返回 NULL 指针。
dynamic_cast 不能用于将非多态基类的指针或引用，强制转换为派生类的指针或引用

dynamic_cast

```
1  class Base { //有虚函数，因此是多态基类
2  public:
3    virtual ~Base() { }
4  };
5  class Derived:public Base  {  };
6  int main() {
7    Base b;
8    Derived d;
9    Derived * pd;
10   pd = reinterpret_cast<Derived*> (&b);
11   if (pd == NULL)
12   //此处 pd 不会为 NULL。reinterpret_cast 不检查安全性，总是进行转换
13     cout << "unsafe reinterpret_cast" << endl; //不会执行
14   pd = dynamic_cast<Derived*> (&b);
15   if (pd == NULL)
16      //结果会是 NULL，因为 &b 不是指向派生类对象，此转换不安全
17     cout << "unsafe dynamic_cast1" << endl;  //会执行
18   pd = dynamic_cast<Derived*> (&d); //安全的转换
19   if (pd == NULL)  //此处 pd 不会为 NULL
20     cout << "unsafe dynamic_cast2" << endl; //不会执行
21   return 0;
22 }
```

程序运行发生异常

程序运行中总难免发生错误

- 数组元素的下标超界、访问 NULL 指针
- 除数为 0
- 动态内存分配 new 需要的存储空间太大
- ...

程序运行发生异常

程序运行中总难免发生错误

- 数组元素的下标超界、访问 NULL 指针
- 除数为 0
- 动态内存分配 new 需要的存储空间太大
- ...

引起这些异常情况的原因：

- 代码质量不高，存在 BUG
- 输入数据不符合要求
- 程序的算法设计时考虑不周到
- ...

程序运行发生异常

发生异常怎么办

- 不只是简单地终止程序运行
- 能够反馈异常情况的信息：哪一段代码发生的、什么异常
- 能够对程序运行中已发生的事情做些处理：取消对输入文件的改动、释放已经申请的系统资源……

异常处理

一个函数运行期间可能产生异常。在函数内部对异常进行处理未必合适。因为函数设计者无法知道函数调用者希望如何处理异常。

告知函数调用者发生了异常，让函数调用者处理比较好

用函数返回值告知异常不方便

用 try、catch 进行异常处理

```cpp
#include <iostream>
using namespace std;
int main() {
  double m ,n;
  cin >> m >> n;
  try {
    cout << "before dividing." << endl;
    if( n == 0)
      throw -1; //抛出 int 类型异常
    else
      cout << m / n << endl;
    cout << "after dividing." << endl;
  }
  catch(double d) {
    cout << "catch(double) " << d <<  endl;
  }
  catch(int e) {
    cout << "catch(int) " << e << endl;
  }
  cout << "finished" << endl;
  return 0;
}
```

捕获任何异常的 catch 块

```
1  int main() {
2    double m ,n;
3    cin >> m >> n;
4    try {
5      cout << "before dividing." << endl;
6      if (n == 0)
7        throw -1; //抛出整型异常
8      else if( m == 0 )
9        throw -1.0; //抛出 double 型异常
10     else
11       cout << m / n << endl;
12     cout << "after dividing." << endl;
13   }
14   catch(double d) {
15     cout << "catch(double) " << d <<  endl;
16   }
17   catch(...) {
18     cout << "catch(...) " << endl;
19   }
20   cout << "finished" << endl;
21   return 0;
22 }
```

异常的再抛出

如果一个函数在执行的过程中，抛出的异常在本函数内就被 catch 块捕获并处理了，那么该异常就不会抛给这个函数的调用者 (也称“上一层的函数”)；如果异常在本函数中没被处理，就会被抛给上一层的函数。

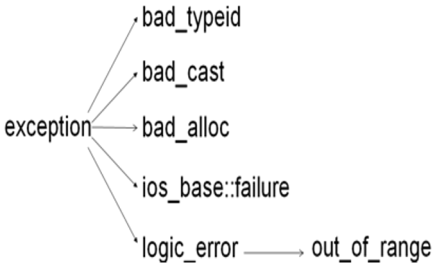
```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class CException{
5  public :
6    string msg;
7    CException(string s):msg(s) { }
8  };
9  double Devide(double x, double y) {
10   if(y == 0)
11     throw CException("devided by zero");
12   cout << "in Devide" << endl;
13   return x / y;
14 }
```

异常的再抛出

```
15  int CountTax(int salary) {
16    try {
17      if( salary < 0 )
18        throw -1;
19        cout << "counting tax" << endl;
20    }
21    catch (int ) {
22      cout << "salary < 0" << endl;
23    }
24    cout << "tax counted" << endl;
25    return salary * 0.15;
26  }
27  int main() {
28    double f = 1.2;
29    try {
30      CountTax(-1);
31      f = Devide(3,0);
32      cout << "end of try block" << endl;
33    }
34    catch(CException e) {
35      cout << e.msg << endl;
36    }
37    cout << "f=" << f << endl;
38    cout << "finished" << endl;
39    return 0;
40  }
```

C++ 标准异常类

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来



bad_cast

在用 dynamic_cast 进行从多态基类对象（或引用），到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常。

```
1  #include <iostream>
2  #include <stdexcept>
3  #include <typeinfo>
4  using namespace std;
5  class Base {
6    virtual void func(){}
7  };
8  class Derived : public Base {
9  public:
10   void Print() { }
11 };
12 void PrintObj( Base & b) {
13   try  {
14     Derived & rd = dynamic_cast<Derived&>(b);
15     //此转换若不安全，会抛出 bad_cast 异常
16     rd.Print();
17   }
18   catch (bad_cast& e) {
19     cerr << e.what() << endl;
20   }
21 }
22 int main () {
23   Base b;
24   PrintObj(b);
25   return 0;
26 }
```

bad_alloc

在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常。

```
1  #include <iostream>
2  #include <stdexcept>
3  using namespace std;
4  int main () {
5    try  {
6      char * p = new char[0x7fffffff];
7    //如果无法分配这么多空间，会抛出异常
8    }
9    catch (bad_alloc & e)  {
10     cerr << e.what() << endl;
11   }
12   return 0;
13 }
```

out_of_range

用 vector 或 string 的 at 成员函数根据下标访问元素时，如果下标越界，就会抛出此异常。例如：

```cpp
#include <iostream>
#include <stdexcept>
#include <vector>
#include <string>
using namespace std;
int main () {
  vector<int> v(10);
  try {
    v.at(100)=100;  //抛出 out_of_range 异常
  }
  catch (out_of_range& e) {
    cerr << e.what() << endl;
  }
  string s = "hello";
  try {
    char c = s.at(100); //抛出 out_of_range 异常
  }
  catch (out_of_range& e) {
    cerr << e.what() << endl;
  }
  return 0;
}
```

输出结果：

```
vector::_M_range_check: __n (which is 100) >= this->size() (which is 10)
basic_string::at: __n (which is 100) >= this->size() (which is 5)
```

运行时类型检查

C++ 运算符 typeid 是单目运算符，可以在程序运行过程中获取一个表达式的值的类型。typeid 运算的返回值是一个 type_info 类的对象，里面包含了类型的信息。

typeid 和 type_info 用法示例

```cpp
#include <iostream>
#include <typeinfo> //要使用 typeinfo，需要此头文件
using namespace std;
struct Base { };    //非多态基类
struct Derived : Base { };
struct Poly_Base {virtual void Func(){ } }; //多态基类
struct Poly_Derived: Poly_Base { };
int main() {
  //基本类型
  long i;  int * p = NULL;
  cout << "1) int is: " << typeid(int).name() << endl; //输出 1) int is: int
  cout << "2) i is: " << typeid(i).name() << endl; ^^I    //输出 2) i is: long
  cout << "3) p is: " <<  typeid(p).name() << endl;     //输出 3) p is: int *
  cout << "4) *p is: " <<  typeid(*p).name() << endl ;  //输出 4) *p is: int
  //非多态类型
  Derived derived;
  Base* pbase = &derived;
  cout << "5) derived is: " << typeid(derived).name() << endl; //输出 5) derived is: Derived
  cout << "6) *pbase is: " << typeid(*pbase).name() << endl;  //输出 6) *pbase is: Base
  cout << "7) " << (typeid(derived)==typeid(*pbase) ) << endl; //输出 7) 0
  //多态类型
  Poly_Derived polyderived;
  Poly_Base* ppolybase = &polyderived;
  cout << "8) polyderived is: " << typeid(polyderived).name() << endl;
  //输出 8) polyderived is: Poly_Derived
  cout << "9) *ppolybase is: " << typeid(*ppolybase).name() << endl;
  //输出 9) *ppolybase is: Poly_Derived
  cout << "10) " << (typeid(polyderived)!=typeid(*ppolybase) ) << endl; ^^I  //输出 10) 0
  return 0;
}
```

多文件共享全局变量

在一个文件中定义，其他文件中用 extern 声明

```
1  //a.cpp:
2  int x = 100;
3
4  //b.cpp:
5  extern int x;  //此处不可初始化
6
7  //c.cpp:
8  extern int x; //此处不可初始化
```

多文件共享全局常量

在一个文件中用 extern const 定义，其他文件中用 extern const 声明

```
1  //a.cpp:
2
3  extern const int x = 100;
4
5  //b.cpp:
6
7  extern const int x;
```

多文件各自使用全局常量：

```
1  //a.cpp:
2
3  const int x = 100;
4
5  // b.cpp:
6
7  const int x = 200;
8
```

指向数组的指针

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main() {
5    int a[3][4];
6    int (*p) [4];  //和 int * p[4] 不同
7    //int (*p) [4]; 一个指针, 它指向一个有 4 个 int 元素的数组
8    //int * p[4]; 表示 p 是一个具有 4 个 int 类型指针的数组。
9    int c[5];
10   int d[4];
11   //p = c; //error
12   p = &d;
13   (*p)[0] = 122;
14   cout << d[0] << endl;  //=> 122
15   p = a;
16   (*p)[0] = 333;
17   cout << a[0][0] << endl; //=> 333
18   p = & a[2] ;
19   (*p)[0] = 444;
20   cout << a[2][0] << endl;  //=>444
21
22   return 0;
23 }
```

指向数组的指针

```
1  typedef int int_4array[4];//int_4array 代表具有 4 个元素的整型数组
2  int main() {
3    int a[3][4];
4    int_4array *p = a;
5    (*p)[0] = 122;
6    a[1][0] = 555;
7    cout << a[0][0] << endl;
8    //输出 a 全部元素
9    for (int_4array *q = a; q != a + 3; ++ q )
10     for(int * s = *q; s != *q + 4; ++s)
11       cout << * s << endl;
12   return 0;
13 }
```

多继承

一个类可以有多个直接基类，这叫多继承

```
1  class A { };
2  class B { };
3  class C :public A, public B { };
4
```

C 类拥有 A,B 类的全部成员

多继承

一个类可以有多个直接基类，这叫多继承

```
1  class A { };
2  class B { };
3  class C :public A, public B { };
4
```

C 类拥有 A,B 类的全部成员
为什么需要多继承?

多继承

一个类可以有多个直接基类，这叫多继承

```
1  class A { };
2  class B { };
3  class C :public A, public B { };
4
```

C 类拥有 A,B 类的全部成员

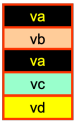
为什么需要多继承?

iostream

多继承的二义性

```
1  class A {
2  public :
3    int va;
4    void fun() { }
5   };  //sizeof(A)=4
6  class B : public  A  {  int vb;    };
7  // sizeof(B) = 8
8  class C :public A    {  int vc;    };
9  // sizeof(C) = 8
10 class D : public B, public C {  int  vd ; };
11 //sizeof( D) =20
12 //.....
13 D d;
14 d.va = 0  //二义性
15 d.B::va = 0;  // ok
16 d.fun(); // 二义性
17 d.C::fun();  // ok
```

对象d的存储结构



virtual 基类 - 避免二义性

在多继承结构中，当派生类 D 的直接基类 parentA 和直接基类 parentB 都基类 base 的派生类时，需要在 parentA 和 parentB 的定义把 base 声明成 virtual 基类，这样在 D 中只有一份从 base 继承的成员。在创建 D 的对象时，

- 由 D 负责初始化 virtual 基类 base
- D 在初始化直接基类 parentA、直接基类 parentB 时，都不进行基类 base 的初始化

```
1  class B : virtual public  A  {  int vb;    };
2  // A 是 B 的虚拟基类， sizeof(B) = 12
3  class C : virtual public A    {  int vc;    };
4  // A 是 C 的虚拟基类， sizeof(C) = 12
5  class D : public B, public C {  int  vd ; };
6  //sizeof( D) =24
7  //.....
8  D d;
9  d.va = 0;  // OK
10 d.fun(); // OK
```

virtual 基类 - 避免二义性

在创建一个派生类的对象时，多继承关系中虚基类的构造函数、析构函数只被执行一次

```
1  class Base {
2  public:
3    int val;
4    Base() { cout << "Base Constructor" << endl; }
5    ~Base() {
6      cout << "Base Destructor" << endl;
7    }
8  };
9  class Base1:virtual public Base {  };
10 class Base2:virtual public Base {  };
11 class Derived:public Base1, public Base2 {   };
12
13 int main() {
14   Derived d;
15   return 0;
16 }
```

virtual 基类 - 避免二义性

在创建一个派生类的对象时，多继承关系中虚基类的构造函数、析构函数只被执行一次

```
1  class Base {
2  public:
3    int val;
4    Base() { cout << "Base Constructor" << endl; }
5    ~Base() {
6      cout << "Base Destructor" << endl;
7    }
8  };
9  class Base1:virtual public Base {  };
10 class Base2:virtual public Base {  };
11 class Derived:public Base1, public Base2 {   };
12
13 int main() {
14   Derived d;
15   return 0;
16 }
```

输出结果：

Base Constructor

Base Destructor

virtual 基类 - 避免二义性

在创建一个派生类的对象时，多继承关系中虚基类的构造函数、析构函数只被执行一次

```
1  class Base {
2  public:
3    int val;
4    Base() { cout << "Base Constructor" << endl; }
5    ~Base() {
6      cout << "Base Destructor" << endl;
7    }
8  };
9  class Base1:virtual public Base {  };
10 class Base2:virtual public Base {  };
11 class Derived:public Base1, public Base2 {   };
12
13 int main() {
14   Derived d;
15   return 0;
16 }
```

输出结果：

Base Constructor

Base Destructor

注意：要避免二义性，要在每条继承路径上都使用虚拟继承

C++ 程序调用 C 函数

声明 C 函数时要在前面加 extern "C"，否则会因名字变异问题，在链接时找不到函数。如：

```
extern "C" void c_func();
```

或将函数声明放入：

```
extern "C" {
  void c_func();
  void c_func2();
}
```

C++ 程序调用 C 函数

声明 C 函数时要在前面加 extern "C"，否则会因名字变异问题，在链接时找不到函数。如：

```
1  //a.c
2  // gcc --shared -o liba.so a.c
3  #include <stdio.h>
4  void c_func() {
5    printf("c_func\n");
6  }
```

```
1  //b.cpp:
2  //g++ b.cpp -L./ -la -o b
3  #include <iostream>
4  extern "C"   void c_func();
5  int main(int argc, char *argv[]) {
6    c_func();
7    return 0;
8  }
```

C 程序调用 C++ 函数

```cpp
// hello.cpp
//g++ -shared -o libhello.so hello.cpp
#include <iostream>
#include <string>
void hello(std::string s) {
  std::cout << s << std::endl;
}
```

```cpp
// mid.cpp
//g++ --shared -o libmid.so mid.cpp -L./ -lhello
#include <iostream>
#include <string>
void hello(std::string s);
extern "C" {
  void m_hello(char *a) {
    std::string s(a);
    hello(s);
  }
}
```

```cpp
// test.c
//gcc -L./  -lmid test.c -o test
#include <stdio.h>
int main() {
  char s[] = "hello";
  m_hello(s);
  return 0;
}
```

Python 程序调用 C 函数

```cpp
//c.cpp
//g++ -o libc.so -shared c.cpp
#include <iostream>
using namespace std;

extern "C" void cpp_func1(const char * s) {
  cout << "cpp_func1:" << s << endl;
}
```

```python
#t.py
import ctypes
from ctypes import * # c 类型库
import struct

libc = CDLL('libc.so') # 装入动态链接库
libc.cpp_func1(c_char_p(bytes("this 高达",encoding="utf-8")));
```

Python 程序调用 C 函数

C Type	Python Type	ctypes Type
char	1-character string	c_char
wchar_t	1-character Unicode string	c_wchar
char	int/long	c_byte
char	int/long	c_ubyte
short	int/long	c_short
unsigned short	int/long	c_ushort
int	int/long	C_int
unsigned int	int/long	c_uint
long	int/long	c_long
unsigned long	int/long	c_ulong
long long	int/long	c_longlong
unsigned long long	int/long	c_ulonglong
float	float	c_float
double	float	c_double
char * (NULL terminated)	string or none	c_char_p
wchar_t * (NULL terminated)	unicode or none	c_wchar_p
void *	int/long or none	c_void_p

名字空间

在多个程序员合作一个大型的 C++ 程序时，一个程序员起的某个全局变量名、类名，有可能和其他程序员起的名字重名。编写大型程序，可能需要使用多个其他公司开发的类库或函数库，如果这些类库和函数库设计的时候都不考虑重名问题，那么同时使用两个不同的类库或函数库产品时，就会碰到无法解决的重名错误。

名字空间

整个程序有一个全局名字空间
可以自定义名字空间

```
namespace 名字空间名 {
    程序片段
}
```

名字空间

```
namespace group1 {
  class A { };
  //......
}
//......
namespace group1 {
  class B { };
  //......
}
```

那么，A,B 都属于名字空间 group1。

名字空间

```
namespace graphics {
  class A { }; // A 属于名字空间 graphics
}
int main() {
  A a;            //编译出错，A 没有定义
  graphics::A b; //OK，指名了 A 所属的名字空间
  return 0;
}
```

那么，A,B 都属于名字空间 group1。

名字空间

用 using namespace XXXX;

```
#include <iostream>
using namespace std;
namespace graphics {
  class A { };
}
using namespace graphics;
int main() {
  A a; //编译没问题,graphics 已覆盖此处
  return 0;
}
```

那么，A,B 都属于名字空间 group1。

名字空间

用 using XXX::YYY; 使得以后的 YYY 都来自名字空间 XXX

```
1  #include <iostream>
2  #include <vector>
3  using std::cout;
4  using std::vector;
5  using std::endl;
6  int main() {
7    vector<int> v; //前面交待过，vector 是属于 std 的
8    vector<int>::iterator i = v.begin();
9    cout << "Hello" << endl; //前面交待过，cout 和 endl 是属于 std 的
10   cout << "World" << endl;
11   return 0;
12 }
```

条件编译

形式一：

```
#ifdef XXX
  //代码块 ，可以由任意内容构成
#endif

// 若前面 #define 了 XXX，则会编译 “代码块”，否则不会
```

```
int main() {
#ifdef XXX
  cout << "hello" <<endl;
#endif
  return 0;
} //无输出
```

```
#define XXX
int main()  {
#ifdef XXX
  cout << "hello" <<endl;
#endif
  return 0;
} //输出 hello
```

条件编译

形式二：

```
#ifdef XXX
  //代码块 1
#else
  //代码块 2
#endif
```

```


#include <iostream>
using namespace std;
#define LINUX
int main()  {
#ifdef LINUX
  cout << "LINUX" <<endl;
#else
  cout << "WINDOWS" <<endl;
#endif
  return 0;
} //输出 LINUX
```

条件编译

```
1
2
3   #undef XXX
4   //取消前面对 XXX 的 #define
5
6   #ifndef XXX
7      // 代码块
8   #endif
9   //XXX 没 define 则编译代码块
10
```

句柄类

要解决的问题：使用基类指针数组存储各派生类对象，要管理 new 出来的对象，比较麻烦。

- 解决办法 1：使用 shared_ptr<CShape> 数组
- 解决办法 2：使用“句柄类”。类似于解决办法 1，自己实现以加深理解。

用“句柄类”实现几何形体程序

```
1  class CShape {
2  public:
3    virtual double Area() const = 0; //纯虚函数
4    virtual void PrintInfo() const = 0;
5    virtual CShape* Clone() const = 0;
6    virtual ~CShape() { };
7  };
8  class CRectangle:public CShape {
9  public:
10   int w,h;
11   virtual double Area() const ;
12   virtual void PrintInfo() const ;
13   virtual CRectangle* Clone() const {
14     return new CRectangle(*this);
15   }
16 };
17 class CCircle:public CShape  {
18 public:
19   int r;
20   virtual double Area() const;
21   virtual void PrintInfo() const;
22   virtual CCircle* Clone() const {
23     return new CCircle(*this);
24   }
25 };
```

用“句柄类”实现几何形体程序

```
26  class CTriangle:public CShape  {
27  public:
28    int a,b,c;
29    virtual double Area() const;
30    virtual void PrintInfo() const;
31    virtual CTriangle* Clone() const {
32      return new CTriangle(*this);
33    }
34  };
35  double CRectangle::Area() const {
36    return w * h;
37  }
38  void CRectangle::PrintInfo()  const {
39    cout << "Rectangle:" << Area() << endl;
40  }
41  double CCircle::Area() const {
42    return 3.14 * r * r ;
43  }
44  void CCircle::PrintInfo() const {
45    cout << "Circle:" << Area() << endl;
46  }
47  double CTriangle::Area() const  {
48    double p = (a + b + c) / 2.0;
49    return sqrt(p * (p - a)*(p - b)*(p - c));
50  }
51  void CTriangle::PrintInfo() const {
52    cout << "Triangle:" << Area() << endl;
53  }
```

用“句柄类”实现几何形体程序

```
54  class Handle {  //句柄类
55  private:
56    int *use;  //指向引用计数的指针
57    CShape *pCShape;  //指向对象的指针
58    void DecreseUse() {
59      if (--*use == 0) {
60        pCShape->PrintInfo();
61        delete pCShape;
62      }
63    }
64  public:
65    Handle() : pCShape(NULL), use(new int(1)) { }
66    Handle(const CShape &item) : pCShape(item.Clone()), use(new int(1)) { }
67    Handle(const Handle &ref) : pCShape(ref.pCShape), use(ref.use){
68      ++*use;
69    }
70    Handle &operator=(const Handle &right) {
71      ++*(right.use);
72      DecreseUse();
73      pCShape = right.pCShape;
74      use = right.use;
75      return *this;
76    }
77    const CShape *operator->() const {  //-> 是单目运算符
78      if (pCShape) return pCShape;
79      else throw logic_error("unbound Handle!");
80    }
```

用“句柄类”实现几何形体程序

```
81      const CShape &operator* () const{
82        if(pCShape) return *pCShape;
83        else throw logic_error("unbound Handle");
84      }
85      bool operator<(const Handle & h) const  {
86        return (*this)->Area() < h->Area();
87      }
88      ~Handle() {
89        DecreseUse();
90      }
91    };
```

用“句柄类”实现几何形体程序

```
92  vector<Handle> shapes;
93  int main() {
94    int i; int n;
95    CRectangle r; CCircle c; CTriangle t;
96    cin >> n;
97    for( i = 0;i < n;i ++ ) {
98      char ch;
99      cin >> ch;
100     switch(ch) {
101       case 'R':
102         cin >> r.w >> r.h;
103         shapes.push_back(Handle(r));
104         break;
105       case 'C':
106         cin >> c.r;
107         shapes.push_back(Handle(c));
108         break;
109       case 'T':
110         cin >> t.a >> t.b >> t.c;
111         shapes.push_back(Handle(t));
112         break;
113     }
114   }
115   sort(shapes.begin(),shapes.end());
116   for( i = 0;i <n;i ++)
117     shapes[i]->PrintInfo();
118   return 0;
119 }
```