



北京大学
PEKING UNIVERSITY

信息科学技术学院

程序设计实习

郭 炜



北京大学
PEKING UNIVERSITY

C++高级特性



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

类型强制转换



山西绵山

static_cast、interpret_cast、const_cast和dynamic_cast

1.static_cast

static_cast用来进行比较“自然”和低风险的转换，比如整型和实数型、字符型之间互相转换。

static_cast不能来在不同类型的指针之间互相转换，也不能用于整型和指针之间的互相转换，也不能用于不同类型的引用之间的转换。

static_cast示例

```
#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 int *,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;  
}
```

2. 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 示例

```
#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);
// la太长, 只取低32位0x5678abcd拷贝给pa
unsigned int u = reinterpret_cast<unsigned int>(pa);
//pa逐个比特拷贝到u
cout << hex << u << endl; //输出 5678abcd
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
```

输出结果:

200,100

400

5678abcd

}

3. `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：

```
const string s = "Inception";  
string & p = const_cast<string&>(s);  
string * ps = const_cast<string*>(&s);  
// &s的类型是const string *
```

4. 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示例

```
#include <iostream>
#include <string>
using namespace std;
class Base
{ //有虚函数, 因此是多态基类
public:
    virtual ~Base() { }
};
class Derived:public Base { };
int main()
{
    Base b;
    Derived d;
    Derived * pd;
    pd = reinterpret_cast<Derived*> ( &b);
```

```
if( pd == NULL)
//此处pd不会为NULL。reinterpret_cast不检查安全性，总是进行转换
    cout << "unsafe reinterpret_cast" << endl; //不会执行
pd = dynamic_cast<Derived*> ( &b);
if( pd == NULL)
//结果会是NULL，因为 &b不是指向派生类对象，此转换不安全
    cout << "unsafe dynamic_cast1" << endl; //会执行
pd = dynamic_cast<Derived*> ( &d); //安全的转换
if( pd == NULL) //此处pd 不会为NULL
    cout << "unsafe dynamic_cast2" << endl; //不会执行
return 0;
}
```

输出结果:

unsafe dynamic_cast1

```
Derived & r = dynamic_cast<Derived&>(b);
```

那该如何判断该转换是否安全呢？

答案：不安全则抛出异常



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

异常处理



张家界

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

- 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

```
#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;  
}
```

程序运行结果如下:

9 6✓

before dividing.

1.5

after dividing.

finished

捕获任何异常的catch块

```
#include <iostream>
using namespace std;
int main()
{
    double m ,n;
    cin >> m >> n;
    try {
        cout << "before dividing." << endl;
        if( n == 0)
            throw -1; //抛出整型异常
        else if( m == 0 )
            throw -1.0; //抛出double型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
```

```

catch(double d) {
    cout << "catch(double) " << d << endl;
}
catch(...) {
    cout << "catch(...)" << endl;
}
cout << "finished" << endl;
return 0;
}

```

程序运行结果:

9 0 ✓

before dividing.

catch(...)

finished

0 6 ✓

before dividing.

catch(double) -1

finished

注意: try块中定义的局部对象, 发生异常时会析构!

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。

异常再抛出

```
#include <iostream>
#include <string>
using namespace std;
class CException
{
    public :
        string msg;
        CException(string s):msg(s) { }
};
```



```
double Devide(double x, double y)
{
    if(y == 0)
        throw CException("devided by zero");
    cout << "in Devide" << endl;
    return x / y;
}

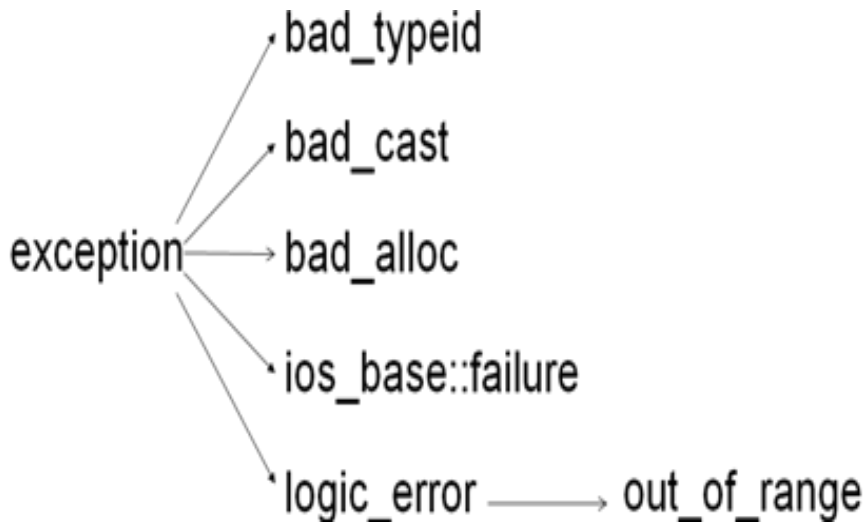
int CountTax(int salary)
{
    try {
        if( salary < 0 )
            throw -1;
        cout << "counting tax" << endl;
    }
    catch (int ) {
        cout << "salary < 0" << endl;
    }
}
```

```
        cout << "tax counted" << endl;
        return salary * 0.15;
    }
int main()
{
    double f = 1.2;
    try {
        CountTax(-1);
        f = Devide(3,0);
        cout << "end of try block" << endl;
    }
    catch(CException e) {
        cout << e.msg << endl;
    }
    cout << "f=" << f << endl;
    cout << "finished" << endl;
    return 0;
}
```

输出结果:
salary < 0
tax counted
devided by zero
f=1.2
finished

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。

```
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;
class Base
{
    virtual void func(){}
};
class Derived : public Base
{
public:
    void Print() { }
};
```

```
void PrintObj( Base & b)
{
    try {
        Derived & rd =
dynamic_cast<Derived>(b);
        //此转换若不安全, 会抛出bad_cast异常
        rd.Print();
    }
    catch (bad_cast& e) {
        cerr << e.what() << endl;
    }
}

int main ()
{
    Base b;
    PrintObj(b);
    return 0;
}
```

输出结果:

Bad dynamic_cast!

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。

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main ()
{
    try {
        char * p = new char[0x7fffffff];
        //无法分配这么多空间，会抛出异常
    }
    catch (bad_alloc & e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

输出结果:

bad allocation

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：

```
#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:

*invalid vector<T> subscript
invalid string position*



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

运行时类型检查



冰岛黄金瀑布

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

```
#include <iostream>
#include <typeinfo> //要使用typeid, 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;
    //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: struct Derived  
cout << "6) *pbase is: " << typeid(*pbase).name() << endl;  
    //输出 6) *pbase is: struct 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: struct Poly_Derived  
cout << "9) *ppolybase is: " << typeid(*ppolybase).name() << endl;  
    //输出 9) *ppolybase is: struct Poly_Derived  
cout << "10) " << (typeid(polyderived)!=typeid(*ppolybase) ) << endl;  
    //输出 10) 0
```

```
}
```

● 实现任意类型的存储

- `any a = 10;` // 存储一个 `int`
- `a = string("hello world");` // 存储字符串
- `a = myclass();` // 存储自己的对象
- `cout << any_cast<int>(a);` // 转换为需要的类型

● 关于 `any_cast`

- 存储的类型与转换的类型不一致会抛出 `bad_any_cast`
- 指针版不一致返回空指针

Boost.Any 成员

```
#include <boost/any.hpp>
```

成员函数:

```
any(); ~any();  
any(const any&);  
any& operator = (const any&);  
any& swap(any&);  
bool empty();  
template<typename ValueType>  
    any(const ValueType&);  
    any& operator = (const ValueType&);
```

独立函数:

```
template<typename ValueType>  
    ValueType any_cast(const any&);    // Throw bad_any_cast  
    const ValueType* any_cast(const any*);    // Return NULL  
    ValueType* any_cast(any*);
```

Boost.Any 示例

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <boost/any.hpp>
using namespace std;
using namespace boost;
class MyClass {    // 后面未经说明MyClass均指此类
public:
    void f() { cout << "func." << endl; }
};
```

```
void print_any(any& a) {  
    int* pi; string* ps; MyClass* pc;  
    if(pi = any_cast<int>(&a))      // 提取对象  
        cout << *pi << endl;  
    else if(ps = any_cast<string>(&a))  
        cout << *ps << endl;  
    else if(pc = any_cast<MyClass>(&a))  
        pc->f();  
}
```



```
int main()
{
    any a = 1;
    cout << any_cast<int>(a) << endl;
    a = string("Hello World!");
    cout << any_cast<string>(a) << endl;
    a = MyClass();
    // Pointer cast.
    any_cast<MyClass>(&a)->f();
    // Use it in a vector.
    vector<any> v;
    v.push_back(100);
    v.push_back(string("Hello World!"));
    v.push_back(MyClass());
    for_each(v.begin(), v.end(), print_any);
}
```

输出:

1

Hello World!

func.

100

Hello World!

func.

Boost.Any

- Any是如何实现的?
 - 抽象基类 + 模板派生 = 编译时的多态
 - class **any_impl**
 - abstract class.
 - template<class T>class **any_impl_t**: public any_impl
 - Implements any_impl.
 - any
 - any_impl* impl;
 - Assign: impl = new any_impl_t(value);
 - Delete: delete impl;
 - RTTI(runtime type identification)
 - #include <typeinfo>
 - const typeid& info = typeid(*p);
 - info.name(), info1 == info2, etc.

```

#include <iostream>
#include <string>
using namespace std;
class MyAny {
public:
    class baseValue { };
    template<class T>
    class MyValue :public baseValue {
    public:
        T value;
        MyValue(const T & v):value(v) { }
        T GetValue() { return value;}
    };
    template <class T>
    MyAny( const T & v):content(new MyValue<T>(v))
    {
        }
    baseValue * content;
};

```

```
template <class T>
T MyAny_cast( const MyAny & m)
{
    return (static_cast<MyAny::MyValue<T> *> (m.content))
        ->value;
}
int main()
{
    MyAny a = 123;
    MyAny b = string("mystring");
    char * p = "Hello";
    MyAny c = p;
    cout << MyAny_cast<int>(a) << endl;
    cout << MyAny_cast<string>(b) << endl;
    cout << MyAny_cast<char *>(c) << endl;
    return 0;
}
```

输出:

123

mystring

Hello

字符串字面值的连接

```
#include <iostream>
using namespace std;
int main()
{
    cout << "this is"
         " a good "
         "thing" << endl;
}
```

输出:

this is a good thing

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

a.cpp:

```
int x = 100;
```

b.cpp:

```
extern int x; //此处不可初始化
```

c.cpp:

```
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

`a.cpp:`

```
extern const int x = 100;
```

`b.cpp:`

```
extern const int x;
```

多文件各自使用全局常量:

a.cpp:

```
const int x = 100;
```

b.cpp:

```
const int x = 200;
```


多文件可重复出现相同inline函数

inline函数写在某头文件，头文件又被多个文件包含

a.cpp:

```
inline void pt()  
{  
    cout << "bad";  
}
```

b.cpp:

```
inline void pt()  
{  
    cout << "bad";  
}
```

多文件可出现同名同参数但内容不同的inline函数

a.cpp:

```
inline void pt()  
{  
    cout << "bad";  
}
```

a.cpp中调用pt输出 "bad"

b.cpp:

```
inline void pt()  
{  
    cout << "good";  
}
```

b.cpp中调用pt则输出 "good"

多文件可包含相同的类定义

a.cpp:

```
class A {  
    int v;  
    public :  
    void print() {  
        cout << "good" << endl;  
    }  
};
```

b.cpp:

```
class A {  
    int v;  
    public :  
    void print() {  
        cout << "good" << endl;  
    }  
};
```

逐行读取输入内容

```
#include <iostream>

#include <string>

using namespace std;

int main()
{
    string s;
    while(getline(cin,s))
        cout << s << endl;    //s中不带换行
    return 0;
}
```

指向数组的指针

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int a[3][4];

    int (*p) [4];    //和 int * p[4]不同

    int c[5];

    int d[4];

    //p = c; //error

    p = &d;

    (*p)[0] = 122;
```

指向数组的指针

```
cout << d[0] << endl; //=> 122
```

```
p = a;
```

```
(*p)[0] = 333;
```

```
cout << a[0][0] << endl; //=> 333
```

```
p = & a[2] ;
```

```
(*p)[0] = 444;
```

```
cout << a[2][0] << endl; //=>444
```

```
return 0;
```

```
}
```

指向数组的指针

```
typedef int int_4array[4];

int main() {
    int a[3][4];
    int_4array *p = a;
    (*p)[0] = 122;
    a[1][0] = 555;
    cout << a[0][0] << endl;
    //输出a全部元素
    for (int_4array *q = a; q != a + 3; ++q)
        for(int * s = *q; s != *q + 4; ++s)
            cout << * s << endl;

    return 0;
}
```

操作数有负数时的%运算

1) 两个操作数都为负数，则结果为负数（或0）

```
-21 % -8 ;    // result is -5
```

2) 两个操作数一正一负

跟机器有关。

如结果符号随分子（被除数），则结果向0一侧取整

```
21 % -5;    //result is 1
```

如果结果符号随分母（除数），则结果向负无穷一侧取整

```
21 % -5;    //result is -4
```


表达式的操作数计算顺序没有规定

`f1 () * f2 ()` 先算 `f1 ()` 还是 `f2 ()` 没规定

```
if( a[index++] < a[index]) { //结果不一定
}
```

基本类型常量不占存储空间

```
#include <iostream>
using namespace std;
const int v = 100;
int main(int argc, char *argv[]) {
    cout << sizeof(v) << endl;
    //int * p = const_cast<int *>(& v);
    int * p = (int *) (&v);
    * p = 200;    //导致程序崩溃
    cout << * p << endl;
    return 0;
}
```

派生类重载的 = 应该调用基类的 =

```
struct A {  
    A & operator = (const A & a) {  
        if( this == & a)  
            return * this;  
        cout << "operator =" << endl;  
        return * this;  
    }  
};  
struct B :public A{  
    B & operator = (const B & b) {  
        if( this != & b)  
            A::operator=(b) ;  
        return * this;  
    }  
};b;
```

const 引用可以绑定到右值以及不同但相关类型的变量

```
#include <iostream>
using namespace std;
int main()
{
    int i = 42;
    const int &r1 = 42;
    const int &r2 = r1 + 10;
    cout << r1 << ", " << r2 << endl;    //=> 42,52
    double d = 3.14;
    const int &rd = d;
    cout << rd << endl;    // => 3
    return 0;
}
```

显式指定函数模板类型参数

```
struct A {  
    A(int b) { }  
};  
  
template <class T1, class T2, class T3>  
T1 Sum(T2 a, T3 b) {  
    return a + b;  
}  
  
int main()  
{  
    Sum<A>(3, 4); // T1 为A。 T2, T3从实参推断  
    Sum<A, int, int>(3, 4);  
    return 0;  
}
```

显式指定函数模板类型参数

```
struct A {  
    A(int b) { }  
};  
template <class T1,class T2,class T3>  
T3 Sum(T1 a,T2 b) {  
    return a + b;  
}  
int main()  
{  
    Sum<A>(3,4); //error  
    Sum<int,int,A>(3,4);  
    return 0;  
}
```

函数指针赋值导致模板实例化

```
template <class T>

int compare(const T & a ,const T & b) {

    return a - b;

}

int main()

{

    typedef int (*PF) (const int & ,const int &);

    PF p = compare;           //导致模板实例化

    p(1,2);

    return 0;

}
```

多继承

一个类可以有多个直接基类，这叫多继承

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

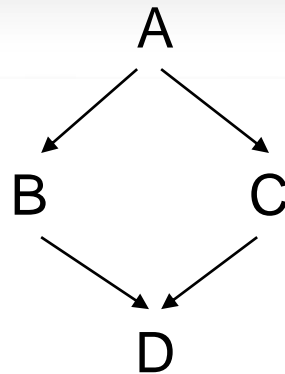
C类拥有A,B类的全部成员

为什么需要多继承？

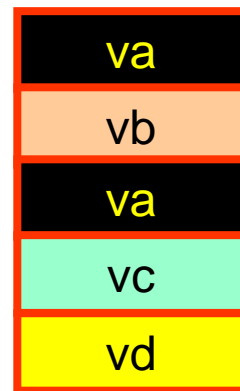
销售经理的例子

多继承的二义性

```
class A {  
    public :  
        int va;  
        void fun() { }  
}; //sizeof(A)=4  
class B : public A { int vb; };  
    // sizeof(B) = 8  
class C :public A  { int vc; };  
    // sizeof(C) = 8  
class D : public B, public C { int vd ; };  
    //sizeof( D) =20  
.....  
D d;  
d.va = 0 //二义性  
d.B::va = 0; // ok  
d.fun(); // 二义性  
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

```
class B : virtual public A { int vb; };  
    // A是B的虚拟基类, sizeof(B) = 12  
class C : virtual public A { int vc; };  
    // A是C的虚拟基类, sizeof(C) = 12  
class D : public B, public C { int vd; };  
    //sizeof( D) =24  
  
.....  
D d;  
d.va = 0; // OK  
d.fun(); // OK
```

- 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

```
class Base {
public:
    int val;
    Base() { cout << "Base Constructor" << endl; }
    ~Base() {
        cout << "Base Destructor" << endl;
    }
};

class Base1:virtual public Base { };
class Base2:virtual public Base { };
class Derived:public Base1, public Base2 { };

main() {
    Derived d;
}
```

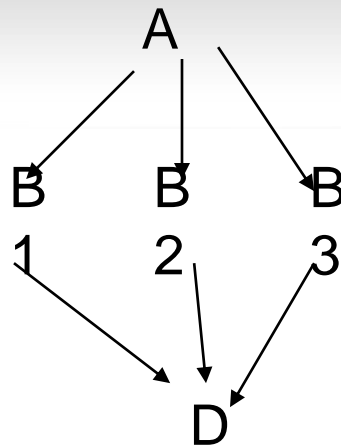
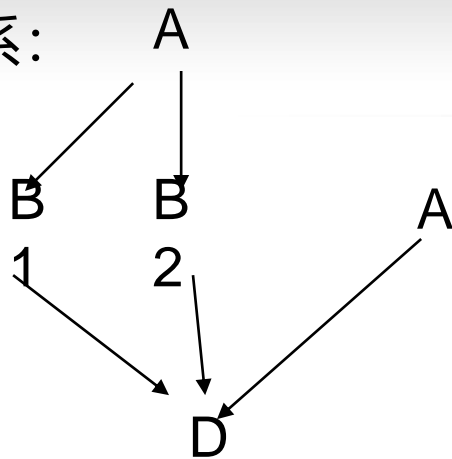
输出结果：
Base Constructor
Base Destructor

注意：要避免二义性，要在每条继承路径上都使用虚拟继承

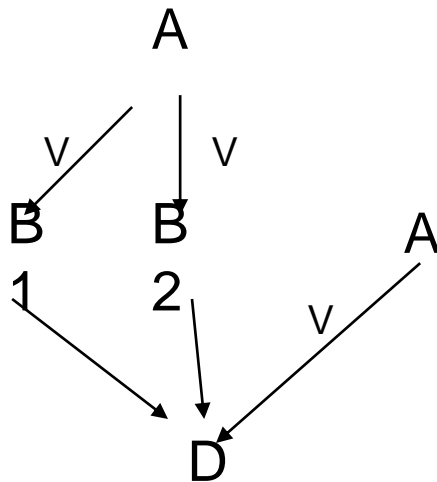
```
class B : virtual public A { int vb ;};    // sizeof(B) = 12  
class C :public A { int vc;};              // sizeof(C) = 8  
class D : public B, public C {int vd ; }; //sizeof( D) = 20
```

```
D d;  
d.va = 0 //二义
```

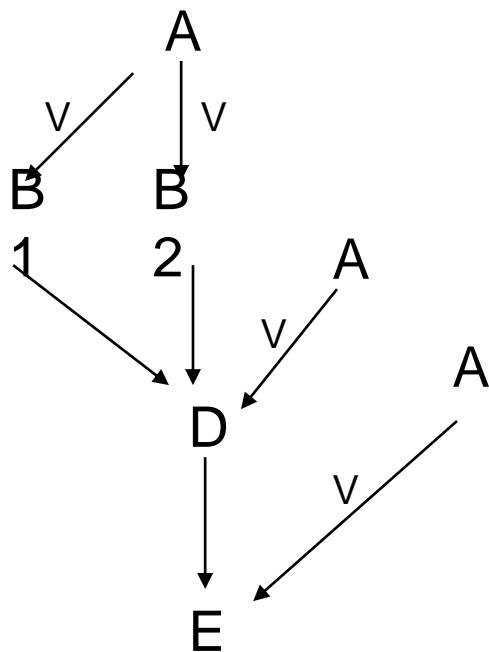
引发二义性的继承关系：



无二义性的继承关系：



正确



多继承派生类对象到基类对象的转换

```
struct A {int n; };  
  
struct B :public A { };  
  
struct C: public A { };  
  
struct D: public B,C { };  
  
int main() {
```

```
    D d;
```

```
    printf("%p\n", &d) ;
```

```
    printf("%p\n", (B*) &d) ;
```

```
    printf("%p\n", (C*) &d) ;
```

```
    printf("%p\n", (A*) &d) ;    // error 二义性
```

```
    A a = d;                    // error 二义性
```

```
    return 0;
```

```
}
```

0152fea8

0152fea8

0152feac

多继承派生类对象到基类对象的转换

```
struct A {      int n;  };

struct B :virtual public A { };

struct C: virtual public A { };

struct D: public B,C { };

int main() {

    D d;

    printf("%p\n", &d) ;

    printf("%p\n", (B*) &d) ;

    printf("%p\n", (C*) &d) ;

    printf("%p\n", (A*) &d) ;

    A a = d;

    return 0;

}
```

0152fea4
0152fea4
0152fea8
0152feac

指向类成员变量的指针

```
struct A {  
    int m;  
    int n;  
    string c;  
    A(int m_,int n_,string c_):m(m_),n(n_),c(c_) { }  
};
```

```
int main()
```

```
{
```

```
    int A::*p1;    //指向A的int 型成员的指针
```

```
    string  A::*p2;    //指向A的string 型成员的指针
```

```
    A a(1,2,"hello");
```

```
    p1 = & A::m;
```

```
    p2 = & A::c;
```

```
    cout << a.*p1 << endl;
```

```
    cout << a.*p2 << endl;
```

```
    p1 = & A::n;
```

```
    cout << a.*p1 << endl;
```

```
    return 0;
```

```
}
```

1
hello
2

指向类成员函数的指针

```
struct A {  
    int n;  
    void func1() const {  
        cout << "func1" << endl;  
    }  
    void func2() const {  
        cout << "func2" << endl;  
    }  
    int func3(int a,int b) {  
        return (a+b)/n;  
    }  
};
```

```
int main() {  
  
    typedef void (A::* PAF1) () const ;  
    typedef int (A::* PAF2) (int ,int );  
    PAF1 p1 = & A::func1;  
  
    A a;  
    (a.*p1) ();  
    p1 = & A::func2;  
    (a.*p1) ();  
    PAF2 p2 = &A::func3;  
    a.n = 2;  
    cout << (a.*p2) (3,5) << endl;  
    return 0;  
  
}
```

func1
func2
4

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();  
}
```

Dev 的 c工程(新建工程时指定C工程) cprj.dev, 内含

a.c

```
#include <stdio.h>
```

```
void c_func()
```

```
{
```

```
    printf("c_func\n");
```

```
}
```

编译得到: a.o

C++程序调用C函数

Dev 的 C++工程 cplus.dev, 内含

main.cpp:

并且在 Project|Project Options|Parameters|Linker 添加 a.o

```
#include <iostream>
```

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

```
int main(int argc, char *argv[]) {
```

```
    c_func();
```

```
    return 0;
```

```
}
```

运行cplusprj.exe 结果: c_func

若无 extern "C" , 则链接时报找不到 ???c_func?? 错

C程序调用C++函数

Dev 的 C++ Static Library 工程 cplusplus2.dev, 内含

c.cpp

```
#include <iostream>

using namespace std;

extern "C" void cpp_func1() { cout << "cpp_func1" << endl; }

void cpp_func1(double m) {
    //再加 extern "C" 会导致函数名冲突编译错
    cout << "cpp_func1:" << m << endl;
}

void cpp_func1(int a, int b) {
    cout << "cpp_func1,a,b=" << a << "," << b<< endl;
}
```


C程序调用C++函数

使用dumpbin 查看 .a 文件, 发现函数发生名字变异

```
dumpbin /all cplusplus.a:
```

```
.... 3 public symbols
```

7E _cpp_func1	//对应 cpp_func1
7E __Z9cpp_func1d	//对应 cpp_func1(double)
7E __Z9cpp_func1ii	//对应 cpp_func1(int,int)

```
.....
```

C程序调用C++函数

Dev 的 C 工程 cprj.dev, 内含

a.c

并且在 Project|Project Options|Parameters|Linker 添加 cplusplusprj2.a

```
#include <stdio.h>
```

```
void cpp_func1();
```

```
void _Z9cpp_func1d(double) ; //名字变异
```

```
void _Z9cpp_func1ii(int ,int); //名字变异
```

```
int main() {
```

```
    cpp_func1();
```

```
    _Z9cpp_func1d(2.98);
```

```
    _Z9cpp_func1ii(1,2);
```

```
    return 0;
```

期待运行结果:

cpp_func1

cpp_func1:2.98

cpp_func1,a,b=1,2

编译时报链接错误:

```
.....  
[Linker error] C:\tmp/c.cpp:7: undefined reference to `std::cout'  
.....
```

编译时报链接错误：

```
.....  
[Linker error] C:\tmp/c.cpp:7: undefined reference to `std::cout'  
.....
```

原因：

cplusprj2.a 里面没有包含C++标准库函数的可执行指令

编译时报链接错误：

```
.....  
[Linker error] C:\tmp/c.cpp:7: undefined reference to `std::cout'  
.....
```

原因：

cplusprj2.a 里面没有包含C++标准库函数的可执行指令

解决：

在工程cprj.dev里，Project|Project Options|Parameters|Linker 添加C++标准库

D:/Application/Dev-Cpp/MinGW64/lib/gcc/i686-w64-mingw32/6.3.0/libstdc++.a

Python程序调用C++函数

c.cpp

```
#include <iostream>

using namespace std;

struct Student {
    char name[20];
    int age;
    double gpa;
};

extern "C" void printStudent(const Student * s){
    cout << s->name << "," << s->age << "," << s->gpa << endl;
}
```

Python程序调用C++函数

c.cpp

```
extern "C" void cpp_func1(const char * s) {  
    cout << "cpp_func1:" << s << endl;  
}  
  
void cpp_func1(double m) {  
    cout << "cpp_func1:" << m << endl;  
}  
  
void cpp_func1(int a, int b) {  
    cout << "cpp_func1,a,b=" << a << ", " << b << endl;  
}
```

编译成动态链接库c.so, 然后拷贝到调用它的python程序的文件夹下:

g++ -o c.so -shared -fPIC c.cpp

Python程序调用C++函数

t.py

```
import ctypes
```

```
from ctypes import *    # c类型库
```

```
import struct
```

```
libc = CDLL('c.so')    #装入动态链接库
```

```
libc.cpp_func1(c_char_p(bytes("this高达",encoding="utf-8")));
```

```
libc._Z9cpp_func1d(c_double(2.98));
```

```
libc._Z9cpp_func1ii(1,2);
```

cpp_func1:this高达

cpp_func1:2.98

cpp_func1,a,b=1,2

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

Python程序调用C++函数

t.py

```
class A:

    def __init__(self,name,gpa,age):

        self.name = name

        self.age = age

        self.gpa = gpa

    def toPack(self):

        return c_char_p(struct.pack("20sid",self.name.encode("utf-8"),self.age,self.gpa))

a = A("this高达",3.4,21)

libc.printStudent(a.toPack()) # this高达,21,3.4
```

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。

- 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  
}
```

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

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

那么，A,B都属于名字空间group1。

```
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;
}
```

用 `using namespace XXXX;` 让XXX起到覆盖作用:

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

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

```
#include <iostream>
```

```
#include <vector>
```

```
using std::cout;
```

```
using std::vector;
```

```
using std::endl;
```

```
int main()
```

```
{
```

```
    vector<int> v; //前面交待过, vector是属于std的
```

```
    vector<int>::iterator i = v.begin();
```

```
    cout << "Hello" << endl; //前面交待过, cout和endl是属于std的
```

```
    cout << "World" << endl;
```

```
    return 0;
```

```
}
```


名字空间

多程序员合作用名字空间避免重复命名

```
//program group1.h
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
namespace group1
```

```
{ //包含一个全局函数名、一个类名、一个函数模板名、一个类模板名
```

```
    void Func1();
```

```
    class A {
```

```
        public:
```

```
        A() { cout << "group1::A()" << endl;}
```

```
        void Print(); //函数体写在group1.cpp中
```

```
    };
```

名字空间

多程序员合作用名字空间避免重复命名

```
template <class T>
void templateFunc(T a)    {
    cout << a << " in group1::templateFunc()" << endl;
}
template <class T>
class templateCls          {
    vector<T> v;
    public:
        void Append(const T & t);
};
template<class T>
void templateCls<T>::Append(const T & t)    {
    v.push_back(t);
    cout << t << " appended in group1::templateCls::append" << endl;
}
```

名字空间

多程序员合作用名字空间避免重复命名

group2.h和group1.h几乎一样，只是把“group1”改成了“group2”：

```
//program group2.h
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
namespace group2
```

{//包含一个全局函数名、一个类名、一个函数模板名、一个类模板名，这它们和group1里的一样

```
void Func1();
```

```
class A {
```

```
public:
```

```
    A() { cout << "group2::A()" << endl;}
```

```
    void Print();//函数体写在group2.cpp中
```

```
};
```

```
template <class T>
```

```
void templateFunc(T a)
```

```
{
```

```
    cout << a << " in group2::templateFunc()" << endl;
```

```
}
```

多程序员合作用名字空间避免重复命名

```
template <class T>
class templateCls
{
    vector<T> v;
    public:
        void Append(const T & t);
};
template<class T>
void templateCls<T>::Append(const T & t)
{
    v.push_back(t);
    cout << t << " appended in group2::templateCls::append" << endl;
}
}
```

名字空间

多程序员合作用名字空间避免重复命名

group1.cpp里包含了group1.h里Func1函数和A::Print函数的函数体:

```
//program group1.cpp
```

```
#include <iostream>
```

```
#include "group1.h"
```

```
using namespace std;
```

```
namespace group1 // 一个namespace分多段写是可以的
```

```
{
```

```
void Func1() { cout << "group1::Func1" << endl; }
```

```
void A::Print() { cout << "group1::A::Print" << endl; }
```

```
}
```

名字空间

多程序员合作用名字空间避免重复命名

group2.cpp和group1.cpp几乎一样，只是把“group1”改成了“group2”：

```
//program group2.cpp
```

```
#include <iostream>
```

```
#include "group2.h"
```

```
using namespace std;
```

```
namespace group2
```

```
{
```

```
void Func1() { cout << "group2::Func1" << endl; }
```

```
void A::Print() { cout << "group2::A::Print" << endl; }
```

```
}
```

名字空间

多程序员合作用名字空间避免重复命名

```
//program namespacedemo.cpp
```

```
#include "group1.h"
```

```
#include "group2.h"
```

```
using namespace std; //此行写不写都一样
```

```
int main() {
```

```
    group1::Func1();    //输出 group1::Func1
```

```
    group1::A a1;        //输出 group1::A()
```

```
    a1.Print();          //输出 group1::A::Print
```

```
    group1::templateFunc("Hello"); //输出 Hello in group1::templateFunc()
```

```
    group1::templateCls<int> t1;
```

```
    t1.Append(100);      //输出 100 appended in group1::templateCls::Append
```

```
    group2::Func1();    //输出 group2::Func1
```

```
    group2::A a2;        //输出 group2::A()
```

```
    a2.Print();          //输出 group2::A::Print
```

```
    group2::templateFunc("Hello"); //输出 Hello in group2::templateFunc()
```

```
    group2::templateCls<int> t2;
```

```
    t2.Append(100);      //输出 100 appended in group2::templateCls::Append
```

```
}
```

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

`#undef XXX`

取消前面对 XXX的 #define

还可以:

`#ifndef XXX`

代码块

`#endif`

XXX没define则编译代码块

类定义内部的枚举类型

```
class A
{
public:
    enum Color { RED, GREEN, BLUE, BLACK };
    int n;
};
```

比将枚举类型写为全局的，更能体现该类型和类的关系

- `int n = A::RED;`
- `A::Color c;`
- `c = A::RED;`

用“句柄类”实现几何形体程序

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

```
class CShape {
public:
    virtual double Area() const = 0; //纯虚函数
    virtual void PrintInfo() const = 0;
    virtual CShape* Clone() const = 0;
    virtual ~CShape() { };
};

class CRectangle:public CShape {
public:
    int w,h;
    virtual double Area() const ;
    virtual void PrintInfo() const ;
    virtual CRectangle* Clone() const {
        return new CRectangle(*this);
    }
};
```

```
class CCircle:public CShape {
public:
    int r;
    virtual double Area() const;
    virtual void PrintInfo() const;
    virtual CCircle* Clone() const {
        return new CCircle(*this);
    }
};

class CTriangle:public CShape {
public:
    int a,b,c;
    virtual double Area() const;
    virtual void PrintInfo() const;
    virtual CTriangle* Clone() const {
        return new CTriangle(*this);
    }
};
```

```
double CRectangle::Area() const {  
    return w * h;  
}  
  
void CRectangle::PrintInfo() const {  
    cout << "Rectangle:" << Area() << endl;  
}  
  
double CCircle::Area() const {  
    return 3.14 * r * r ;  
}  
  
void CCircle::PrintInfo() const {  
    cout << "Circle:" << Area() << endl;  
}  
  
double CTriangle::Area() const {  
    double p = ( a + b + c ) / 2.0;  
    return sqrt(p * ( p - a)*(p- b)*(p - c));  
}  
  
void CTriangle::PrintInfo() const {  
    cout << "Triangle:" << Area() << endl;  
}
```

```
class Handle {    //句柄类
private:
    int *use;      //指向引用计数的指针
    CShape *pCShape; //指向对象的指针
    void DecreaseUse() {
        if (--*use == 0) {
            pCShape->PrintInfo();
            delete pCShape;
        }
    }
public:
    Handle() : pCShape(NULL), use(new int(1)) { }
    Handle(const CShape &item) : pCShape(item.Clone()),
        use(new int(1)) { }
    Handle(const Handle &ref) : pCShape(ref.pCShape), use(ref.use) {
        ++*use;
    }
}
```



```

Handle &operator=(const Handle &right) {
    ++*(right.use);
    DecreaseUse();
    pCShape = right.pCShape;
    use = right.use;
    return *this;
}

const CShape *operator->() const { //->是单目运算符
    if (pCShape)
        return pCShape;
    else
        throw logic_error("unbound Handle!");
}

const CShape &operator* () const{
    if(pCShape)
        return *pCShape;
    else
        throw logic_error("unbound Handle");
}

```

```
bool operator<(const Handle & h) const {  
    return (*this)->Area() < h->Area();  
}
```

```
~Handle() {  
    DecreseUse();  
}
```

```
};
```

```
vector<Handle> shapes;  
int main() {  
    int i; int n;  
    CRectangle r; CCircle c; CTriangle t;  
    cin >> n;  
    for( i = 0; i < n; i ++ ) {  
        char ch;  
        cin >> ch;  
        switch(ch) {  
            case 'R':  
                cin >> r.w >> r.h;  
                shapes.push_back(Handle(r));  
                break;  
            case 'C':  
                cin >> c.r;  
                shapes.push_back(Handle(c));  
                break;  
        }
```

```
        case 'T':
            cin >> t.a >> t.b >> t.c;
            shapes.push_back(Handle(t));
            break;
    }
}
sort(shapes.begin(), shapes.end());
for( i = 0; i < n; i++)
    shapes[i]->PrintInfo();
return 0;
}
```