



程序设计与算法(三)

C++面向对象程序设计

郭 炜

微博：<http://weibo.com/guoweiofpku>

学会程序和算法，走遍天下都不怕！

讲义照片均为郭炜拍摄



北京大学
PEKING UNIVERSITY

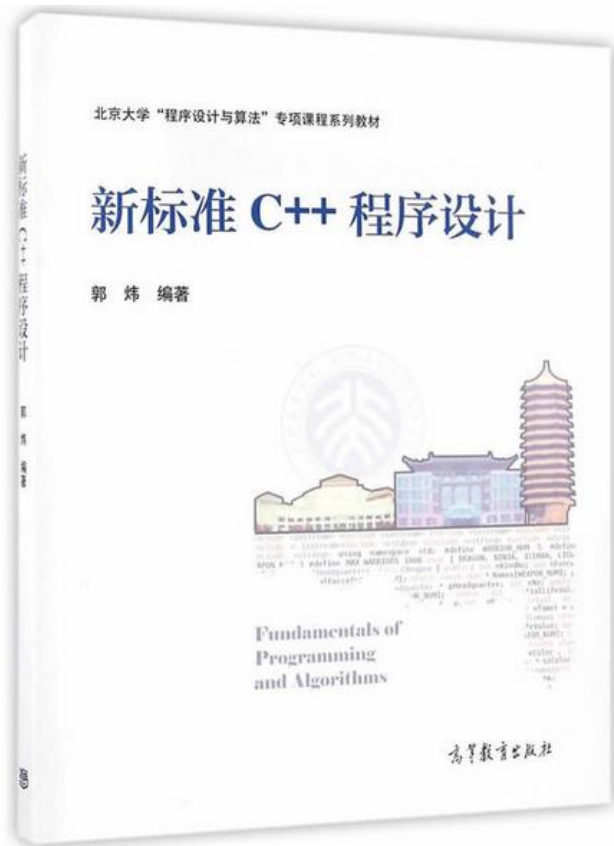
信息科学技术学院

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著





北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

C++11特性



九寨沟

统一的初始化方法

```
int arr[3]{1, 2, 3};  
vector<int> iv{1, 2, 3};  
map<int, string> mp{{1, "a"}, {2, "b"}};  
string str{"Hello World"};  
int * p = new int[20]{1,2,3};
```

```
struct A {  
    int i,j;  A(int m,int n):i(m),j(n) {  
    };  
A func(int m,int n ) { return {m,n}; }  
int main() { A * pa = new A {3,7}; }
```

成员变量默认初始值

```
class B
{
    public:
        int m = 1234;
        int n;
};

int main()
{
    B b;
    cout << b.m << endl;    //输出 1234
    return 0;
}
```

auto关键字

用于定义变量，编译器可以自动判断变量的类型

```
auto i = 100;           // i 是 int
auto p = new A();       // p 是 A *
auto k = 34343LL;       // k 是 long long
```

```
map<string,int,greater<string> > mp;
for( auto i = mp.begin(); i != mp.end(); ++i)
    cout << i->first << "," << i->second ;
//i的类型是： map<string,int,greater<string> >::iterator
```

auto关键字

```
class A { };  
  
A operator + ( int n,const A & a)  
{  
    return a;  
}  
  
template <class T1, class T2>  
auto add(T1 x, T2 y) -> decltype(x + y) {  
    return x+y;  
}  
  
auto d = add(100,1.5); // d是double d=101.5  
auto k = add(100,A()); // d是A类型
```

decltype 关键字

求表达式的类型

```
int i;  
double t;  
struct A { double x; };  
const A* a = new A();
```

```
decltype(a)      x1;    // x1 is A *  
decltype(i)      x2;    // x2 is int  
decltype(a->x)    x3;    // x3 is double  
decltype((a->x))  x4 = t; // x4 is double&
```


智能指针shared_ptr

- 头文件：<memory>
- 通过shared_ptr的构造函数，可以让shared_ptr对象托管一个new运算符返回的指针，写法如下：
- `shared_ptr<T> ptr(new T);` // T 可以是 int ,char, 类名等各种类型
此后ptr就可以像 T* 类型的指针一样来使用，即 *ptr 就是用new动态分配的那个对象，而且不必操心释放内存的事。
- 多个shared_ptr对象可以同时托管一个指针，系统会维护一个托管计数。当无shared_ptr托管该指针时，delete该指针。
- shared_ptr对象不能托管指向动态分配的数组的指针，否则程序运行会出错

智能指针shared_ptr

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

struct A {
    int n;
    A(int v = 0):n(v){ }
    ~A() { cout << n << " destructor" << endl; }
};

int main()
{
    shared_ptr<A> sp1(new A(2)); //sp1托管A(2)
    shared_ptr<A> sp2(sp1);      //sp2也托管 A(2)
    cout << "1)" << sp1->n << ", " << sp2->n << endl;
    //输出1) 2,2
    shared_ptr<A> sp3;
    A * p = sp1.get();          //p 指向 A(2)
    cout << "2)" << p->n << endl;
```

输出结果：

1) 2, 2

2) 2

```

sp3 = sp1; //sp3也托管 A(2)
cout << "3)" << (*sp3).n << endl; //输出 2
sp1.reset(); //sp1放弃托管 A(2)
if( !sp1 )
    cout << "4)sp1 is null" << endl; //会输出
A * q = new A(3);
sp1.reset(q); // sp1托管q
cout << "5)" << sp1->n << endl; //输出 3
shared_ptr<A> sp4(sp1); //sp4托管A(3)
shared_ptr<A> sp5;
sp1.reset(); //sp1放弃托管 A(3)
cout << "before end main" << endl;
sp4.reset(); //sp1放弃托管 A(3)
cout << "end main" << endl;
return 0; //程序结束, 会delete 掉A(2)

```

```

}

```

输出结果：

```

1) 2, 2
2) 2
3) 2
4) sp1 is
null
5) 3
before end
main
3
destructor

```

智能指针shared_ptr

```
#include <iostream>
#include <memory>
using namespace std;
struct A{
    ~A() { cout << "~A" << endl; }
};
int main()
{
    A * p = new A();
    shared_ptr<A> ptr(p);
    shared_ptr<A> ptr2;
    ptr2.reset(p);    //并不增加 ptr中对p的托管计数
    cout << "end" << endl;
    return 0;
}
```

智能指针shared_ptr

```
#include <iostream>
#include <memory>
using namespace std;
struct A{
    ~A() { cout << "~A" << endl; }
};
int main()
{
    A * p = new A();
    shared_ptr<A> ptr(p);
    shared_ptr<A> ptr2;
    ptr2.reset(p);    //并不增加 ptr中对p的托管计数
    cout << "end" << endl;
    return 0;
}
```

输出：

end

~A

~A

之后程序崩溃

因 p被delete两次

空指针nullptr

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

int main()    {
    int* p1 = NULL;
    int* p2 = nullptr;
    shared_ptr<double> p3 = nullptr;
    if(p1 == p2)
        cout << "equal 1" <<endl;
    if( p3 == nullptr)
        cout << "equal 2" <<endl;
    if( p3 == p2) ; // error
    if( p3 == NULL)
        cout << "equal 4" <<endl;
    bool b = nullptr; // b = false
    int i = nullptr; //error,nullptr不能自动转换成整型
    return 0;
}
```

去掉出错的语句后输出：

equal 1

equal 2

equal 4

基于范围的for循环

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

struct A {    int n; A(int i):n(i) {    } };

int main() {
    int ary[] = {1,2,3,4,5};
    for(int & e: ary)
        e*= 10;
    for(int e : ary)
        cout << e << ",";
    cout << endl;
    vector<A> st(ary,ary+5);
    for( auto & it: st)
        it.n *= 10;
    for( A it: st)
        cout << it.n << ",";
    return 0;
}
```

输出：

10,20,30,40,50,
100,200,300,400
,500,

右值引用和move语义

右值：一般来说，不能取地址的表达式，就是右值，
能取地址的，就是左值

```
class A { };
```

```
A & r = A(); // error , A()是无名变量，是右值
```

```
A && r = A(); //ok, r 是右值引用
```

主要目的是提高程序运行的效率，减少需要进行深拷贝的对象进行深拷贝的次数。

[参考](#)

<http://amazingjxq.com/2012/06/06/%E8%AF%91%E8%AF%A6%E8%A7%A3c%E5%8F%B3%E5%80%BC%E5%BC%95%E7%94%A8/>

<http://www.cnblogs.com/soaliap/archive/2012/11/19/2777131.html>


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

class String
{
public:
    char * str;
    String():str(new char[1]) { str[0] = 0;}
    String(const char * s) {
        str = new char[strlen(s)+1];
        strcpy(str,s);
    }
    String(const String & s) {
        cout << "copy constructor called" << endl;
        str = new char[strlen(s.str)+1];
        strcpy(str,s.str);
    }
}
```

```
String & operator=(const String & s) {  
    cout << "copy operator= called" << endl;  
    if( str != s.str) {  
        delete [] str;  
        str = new char[strlen(s.str)+1];  
        strcpy(str,s.str);  
    }  
    return * this;  
}  
  
// move constructor  
String(String && s):str(s.str) {  
    cout << "move constructor called"<<endl;  
    s.str = new char[1];  
    s.str[0] = 0;  
}
```

```
// move assignment
```

```
String & operator = (String &&s) {  
    cout << "move operator= called"<<endl;  
    if (str!= s.str) {  
        delete [] str;  
        str = s.str;  
        s.str = new char[1];  
        s.str[0] = 0;  
    }  
    return *this;  
}  
~String() { delete [] str; }  
};
```

```
template <class T>
void MoveSwap(T& a, T& b)    {
    T tmp(move(a)); // std::move(a) 为右值，这里会调用move
    constructor
    a = move(b); // move(b) 为右值，因此这里会调用move assignment
    b = move(tmp); // move(tmp) 为右值，因此这里会调用move
    assignment
}
```

```
int main()
{
    //String & r = String("this"); // error
    String s;
    s = String("ok"); // String("ok")是右值
    cout << "*****" << endl;
    String && r = String("this");
    cout << r.str << endl;
    String s1 = "hello", s2 = "world";
    MoveSwap(s1, s2);
    cout << s2.str << endl;
    return 0;
}
```

输出：

move operator= called

this

move constructor called
move operator= called
move operator= called
hello

函数返回值为对象时，返回值对象如何初始化？

➤只写复制构造函数

return 局部对象 -> 复制

return 全局对象 -> 复制

➤只写移动构造函数

return 局部对象 -> 移动

return 全局对象 -> 默认复制

return move(全局对象) -> 移动

➤同时写 复制构造函数和 移动构造函数:

return 局部对象 -> 移动

return 全局对象 -> 复制

return move(全局对象) -> 移动

dev c++中，return 局部对象 会导致优化，不调用移动或复制构造函数

可移动但不可复制的对象：

```
struct A{
    A(const A & a) = delete;
    A(const A && a) { cout << "move" << endl; }
    A() { };
};
A b;
A func() {
    A a;
    return a;
}
void func2(A a) { }
int main() {
    A a1;
    A a2(a1);      //compile error
    func2(a1);     //compile error
    func();
    return 0;
}
```

无序容器(哈希表)

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    unordered_map<string,int> turingWinner; //图灵奖获奖名单
    turingWinner.insert(make_pair("Dijkstra",1972));
    turingWinner.insert(make_pair("Scott",1976));
    turingWinner.insert(make_pair("Wilkes",1967));
    turingWinner.insert(make_pair("Hamming",1968));
    turingWinner["Ritchie"] = 1983;
    string name;
    cin >> name; //输入姓名
```



```
        unordered_map<string,int>::iterator p =
turingWinner.find(name);
        //据姓名查获获奖时间
        if( p != turingWinner.end())
            cout << p->second;
        else
            cout << "Not Found" << endl;
        return 0;
    }
```

哈希表插入和查询的时间复杂度几乎是常数

正则表达式

```
#include <iostream>
#include <regex> //使用正则表达式须包含此文件
using namespace std;
int main()
{
    regex reg("b.?p.*k");
    cout << regex_match("bopggk", reg) << endl; //输出 1, 表示匹配成功
    cout << regex_match("boopgggk", reg) << endl; //输出 0, 匹配失败
    cout << regex_match("b pk", reg) << endl; //输出 1, 表示匹配成功
    regex reg2("\\d{3}([a-zA-Z]+). (\\d{2}|N/A)\\s\\1");
    string correct="123Hello N/A Hello";
    string incorrect="123Hello 12 hello";
    cout << regex_match(correct, reg2) << endl; //输出 1, 匹配成功
    cout << regex_match(incorrect, reg2) << endl; //输出 0, 失败
}
```

Lambda表达式

只使用一次的函数对象，能否不要专门为其编写一个类？

只调用一次的简单函数，能否在调用时才写出其函数体？

Lambda表达式

形式：

[外部变量访问方式说明符](参数表) -> 返回值类型

```
{  
    语句组  
}
```

[] 不使用任何外部变量

[=] 以传值的形式使用所有外部变量

[&] 以引用形式使用所有外部变量

[x, &y] x 以传值形式使用，y 以引用形式使用

[=, &x, &y] x, y 以引用形式使用，其余变量以传值形式使用

[&, x, y] x, y 以传值的形式使用，其余变量以引用形式使用

“->返回值类型”也可以没有，没有则编译器自动判断返回值类型。

Lambda表达式

```
int main()
{
    int x = 100,y=200,z=300;
    cout << [ ](double a,double b) { return a + b; }(1.2,2.5)
        << endl;
    auto ff = [=,&y,&z](int n) {
        cout <<x << endl;
        y++; z++;
        return n*n;
    };
    cout << ff(15) << endl;
    cout << y << "," << z << endl;
}
```

Lambda表达式

```
int main()
{
    int x = 100,y=200,z=300;
    cout << [ ](double a,double b) { return a + b; } (1.2,2.5)
        << endl;
    auto ff = [=,&y,&z](int n) {
        cout <<x << endl;
        y++; z++;
        return n*n;
    };
    cout << ff(15) << endl;
    cout << y << ", " << z << endl;
}
```

输出：
3.7
100
225
201,301

Lambda表达式

```
int a[4] = { 4,2,11,33};  
sort(a,a+4,[ ](int x,int y)->bool {  
    return x%10 < y%10; } );  
  
for_each(a,a+4,[ ](int x) {cout << x << " " ;} ) ;
```

Lambda表达式

```
int a[4] = { 4,2,11,33};  
sort(a,a+4,[ ](int x,int y)->bool { return x%10  
                                         < y%10; });  
for_each(a,a+4,[ ](int x) {cout << x << " " ;} ) ;
```

输出：
11 2 33 4

Lambda表达式

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
    vector<int> a { 1,2,3,4};
    int total = 0;
    for_each(a.begin(),a.end(), [&](int & x)
        {total += x; x*=2;});
    cout << total << endl; //输出 10
    for_each(a.begin(),a.end(),[ ](int x)
        { cout << x << " ";});
    return 0;
}
```

程序输出结果：

10

2 4 6 8

Lambda表达式

实现递归求斐波那契数列第n项：

```
function<int(int)> fib = [&fib](int n)
{ return n <= 2 ? 1 : fib(n-1) + fib(n-2);};
```

```
cout << fib(5) << endl;    //输出5
```

`function<int(int)>` 表示返回值为 `int`，有一个`int`参数的函数

多线程

```
#include <iostream>
#include <thread>
using namespace std;
struct MyThread {
    void operator () () {
        while(true)
            cout << "IN MYTHREAD\n";
    }
};
void my_thread(int x)
{
    while(x)
        cout << "in my_thread\n";
}
```

多线程

```
int main()
{
    MyThread x;    // 对x 的要求：可复制
    thread th(x);  // 创建线程并执行
    thread th1(my_thread, 100);
    while(true)
    cout << "in main\n";
    return 0;
}
```

输出：

in my_thread
in my_thread
IN MYTHREAD
IN MYTHREAD
in main
in my_thread
IN MYTHREAD
IN MYTHREAD
in main
in main
in my_thread
IN MYTHREAD
.....



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”和低风险转换，比如整型和实数型、字符型之间互相转换。

static_cast不能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;
}

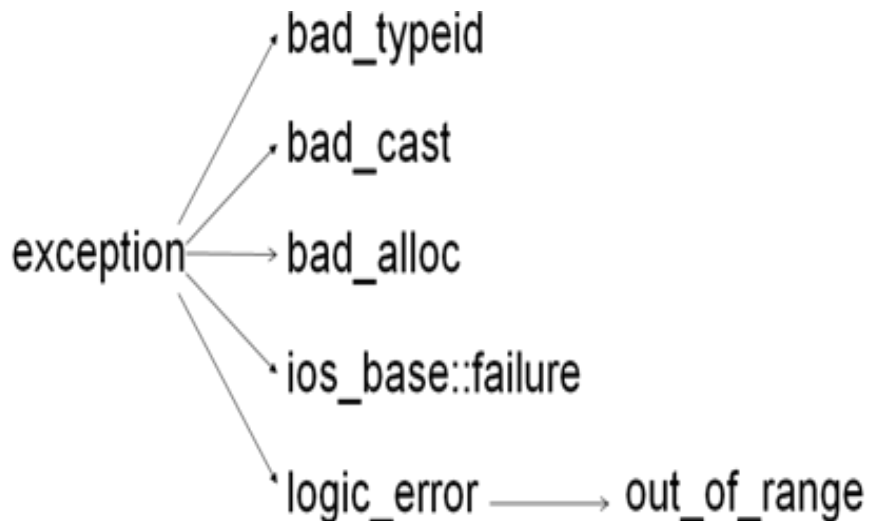
int CountTax(int salary)
{
    try {
        if( salary < 0 )
            throw -1;
        cout << "counting tax" << endl;
    }
    catch (int ) {
        cout << "salary < 0" << endl;
    }
}
```

```
        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
```

```
}
```